Understanding and Detecting Platform-Specific Violations in Android Auto Apps

Moshood Abiola Fakorede
Louisiana State University
Baton Rouge, Louisiana, USA
mfakor1@lsu.edu

Umar Farooq
Louisiana State University
Baton Rouge, Louisiana, USA
ufarooq@lsu.edu

Abstract

Despite over 3.5 million Android apps and 200+ million Android Auto-compatible vehicles, only a few hundred apps support Android Auto due to platform-specific compliance requirements. Android Auto mandates service-based architectures in which the vehicle system invokes app callbacks to render the UI and handle interactions, which is fundamentally different from standard Activity-based Android development. Through an empirical study analysis of 98 issues across 14 Android Auto app repositories, we identified three major compliance failure categories: media playback errors, UI rendering issues, and voice command integration failures in line with mandatory requirements for integrating Android Auto support. We introduce AutoComply, a static analysis framework capable of detecting these compliance violations through the specialized analysis of platform-specific requirements. AutoComply constructs a Car-Control Flow Graph (CCFG) extending traditional control flow analysis to model the service-based architecture of Android Auto apps. Evaluating AutoComply on 31 large-scale open-source apps, it detected 27 violations (13× more than Android Lint), while no false positives were observed, achieving 2× faster analysis. Developers have acknowledged 14 of these violations with 8 fixes already implemented, validating AutoComply's practical effectiveness.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools.*

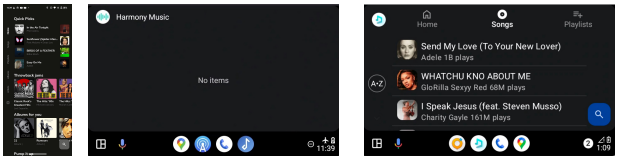
Keywords

Platform Compliance, Android Auto Apps, Static Analysis.

ACM Reference Format:
Moshood Abiola Fakorede and Umar Farooq. 2026. Understanding and Detecting Platform-Specific Violations in Android Auto Apps. In *7th ACM/IEEE International Conference on Automation of Software Test (AST '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3793654.3793745

1 Introduction

Mobile apps play a key role in communication, entertainment, and navigation, extending their influence into the automotive sector through platforms such as Android Auto. Android Auto enables drivers to access mobile apps safely via in-car infotainment systems. Although the Google Play Store lists over 3.5 million apps [7] and more than 200 million vehicles support Android Auto as of 2024 [39], only a small number of apps meet the platform's compliance requirements. As of March 2025, just 363 apps are Android Auto compliant. This disparity points to technical and design challenges that developers face when making their apps compliant for automotive use. Addressing these challenges and understanding the specific compliance barriers is critical for expanding the availability of high-quality, in-vehicle app experiences.



(a) Phone (b) Blank car screen due to platform issue (c) UI appears normally on car screen after fix

Figure 1: Example Android Auto platform issue in an app, (a) app works fine on the phone. (b) Due to compliance issues with Android Auto, the app does not appear as intended on the car screen. (c) After we reported the issue to the developer, the developer was able to fix the problem.

Android Auto platform requirements. In contrast to standard Android apps, Auto apps are built around a service-based architecture. In this model, background services serve as the primary components for handling user interaction, media browsing, and playback, rather than relying on activities.

To be considered compliant with the Android Auto platform, Android apps must meet several additional criteria defined by the Android Auto framework [8]. An app is considered *Android Auto compliant* if it satisfies the following requirements: *(1) Discoverability:* The app must be properly declared in the manifest with appropriate automotive metadata and service registrations to ensure it is discoverable by the system. *(2) Voice Integration:* The app must integrate required voice command APIs to support safe, hands-free interaction. *(3) UI Compliance:* The app must use designated UI templates and construct its content hierarchy through `MediaBrowserService`, ensuring that the interface renders and behaves correctly on in-car displays. *(4) Functional Requirements:* The app must implement category-specific callbacks and other platform-defined behaviors, particularly for media playback functionality.

Figure 1 illustrates a typical Android auto specific we discovered in which the app fails to meet some of these criteria. While the app displays its music library as intended on the phone (a), it fails to render properly on the car screen due to missing callbacks, preventing content rendering in Android Auto (b). After we identified and reported the missing callbacks, the developer implemented them, resolving the issue (c).

AST '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2476-3/26/04
https://doi.org/10.1145/3793654.3793745

This example demonstrates how Android Auto's service-based architecture imposes requirements beyond standard Android development requirements. Such platform-specific requirements are difficult to detect manually and are often missed by existing static analysis tools, motivating our work.

Formative Study. Meeting Android Auto's above-mentioned compliance requirements poses technical and procedural challenges for developers. To better understand these barriers in practice, we conducted a formative study to identify the most frequent and impactful sources of compliance failure and to inform the development of automated methods for detecting and addressing these issues.

To conduct this study, we focus on open-source apps from F-Droid [10] by collecting 4, 387 apps in total. We filter for those with Android Auto tags declared in their manifest files, identifying 37 apps with developers' intent to add Auto support. We subsequently analyzed the repositories of these apps, identifying 98 Android Auto specific issues across 14 repositories. Our analysis reveals that these issues fall primarily into three categories: *(i)* UI compliance issues (as shown in Figure 1), *(ii)* media playback challenges, and *(iii)* failure to implement voice command integration. This comprehensive study provided valuable insights into the technical challenges developers face when adapting their apps for Android Auto's unique requirements.

State of the Art. Recent work [47] has focused on testing message synchronization and UI inconsistencies between an Android phone and a vehicle's screen, but this work primarily addresses messaging functionality and overlooks broader challenges such as UI compliance, media playback, and voice command integration that affect the majority of Android Auto apps. Current static analysis tools fall short of addressing Android Auto's unique requirements. Android Lint [16] only include a single basic rules for voice command integration but lacks comprehensive coverage of Android Auto-specific compliance checks. Methodologically, existing static analysis tools such as FlowDroid [4] rely on interprocedural control-flow graph (ICFG). The ICFG construction misses essential host-driven callbacks, such as `onGetRoot()` and `onPlayFromSearch()`. The Android Auto runtime invokes these callbacks and are critical for platform compliance. This limitation is illustrated in Figure 2, which contrasts ICFG with our augmented Car-Control Flow Graph (CCFG). The CCFG extends the ICFG by introducing synthetic nodes and edges that explicitly model host-invoked and Assistant-driven flows, manifest and service contracts, and UI template population. By making these platform-specific interactions visible to static analysis, the CCFG enables comprehensive compliance checking for Android Auto apps and addresses a key gap left by existing tools.

Overview of AutoComply. In this work, we introduce AutoComply, a static analysis tool that detects a broad range of compliance violations unique to Android Auto apps. At the core AutoComply builds on the Car-Control Flow Graph (CCFG), an analysis abstraction that extends conventional control-flow graphs with host-invoked entry points and platform-specific callbacks. By capturing interactions across app components, lifecycle methods, and Android Auto APIs, the CCFG enables AutoComply to systematically detect compliance violations, including missing or misconfigured callbacks, incomplete voice command support, and UI malfunction, which would otherwise go unnoticed by traditional analysis pipelines.

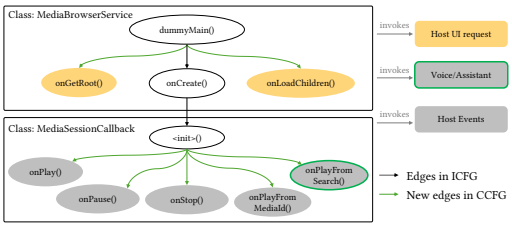


Figure 2: Car Control Flow Graph (CCFG) for Android Auto media apps. Nodes and arrows in black represent standard ICFG code paths; gray nodes indicate required callbacks for UI compliance, and yellow nodes indicate required callbacks for media or voice compliance. Green arrows illustrate host-driven edges added in the CCFG: the Host UI request invokes UI callbacks (`onGetRoot()`, `onLoadChildren()`), while Host Events and Voice/Assistant components invoke the required media and voice command callbacks (`onPlay()`, `onPause()`, `onStop()`, `onPlayFromMediaId()`, and `onPlayFromSearch()`).

Building on the CCFG, AutoComply implements three dedicated compliance checkers; each checker addresses an important source of platform failures in Android Auto apps found in our study. The Media Checker examines the correct handling of playback actions and resource management, focusing on callbacks such as `onPlay()`, `onPause()`, and `onStop()`. The UI Checker verifies that required callbacks for UI population (such as `onGetRoot()` and `onLoadChildren()`) are correctly implemented, ensuring that media catalogs and navigation menus appear as intended on vehicle displays. The Voice Command Checker assesses support for hands-free operation by analyzing Assistant-driven callbacks, most notably `onPlayFromSearch()`. Together, these checkers provide a comprehensive coverage of Android Auto's core compliance requirements, leveraging the precision of the CCFG to highlight missing or faulty implementations.

To evaluate AutoComply, we applied it to a diverse set of 31 Android Auto apps and detected 27 compliance issues, identifying 13X more issues than Android Lint [16] (baseline tool) while no false positives were observed. The tool efficiently analyzes apps of varying complexity, demonstrating its applicability. Additionally, we reported the detected issues to developers, 14 problems acknowledged, and 8 of them have been fixed by the developers so far. By accurately identifying real-world compliance violations, AutoComply provides developers with actionable insights to improve platform compliance and the in-vehicle user experience. The artifact is available at: https://github.com/fakorede/autocomply.

Contributions. To summarize, this work makes the following key contributions:

- We present the first comprehensive study focused on understanding the challenges faced by developers in creating Android Auto-compliant apps.
- We propose Car Control Flow Graph (CCFG), which models the control flow specific to Android Auto apps based on the available features in the apps.
- We developed a novel static analysis specifically designed to identify compliance issues in Android Auto apps, including UI issues and problems with media playback and voice command integration.

- We evaluate AutoComply on real-world apps, showing its effectiveness by detecting 27 issues related to Android Auto compliance, 14 of which were confirmed by developers, and 8 of them have already been fixed.

2 Formative Study

To understand the challenges app developers encounter when adapting apps for Android Auto, we conducted a formative study analyzing real-world issues reported in open-source Android Auto app repositories. This study aims to answer:

RQ1: *What are the most prevalent types of Android Auto compliance violations, and what platform-specific factors contribute to their occurrence?*

Our findings inform the design of AutoComply by identifying the specific compliance checks needed to detect violations that existing tools miss.

2.1 Study Corpus Construction

Data Collection. We collected Android apps from F-Droid [10], an open-source Android app repository, resulting in 4, 387 apps. We selected F-Droid because (i) it hosts a large collection of actively maintained open-source Android apps, and (ii) the apps include publicly accessible source code on platforms like GitHub [13] and GitLab [14] with issue tracking systems, enabling detailed analysis of reported problems and their fixes.

Android Auto App Filtering. To identify apps with Android Auto support, we analyzed each app's `AndroidManifest.xml` file for Android Auto-specific declarations. Apps must declare both automotive metadata and `MediaBrowserService` registration to support Android Auto, as shown in Listing 1.

```
<meta-data
    android:name="com.google.android.gms.car.application"
    android:resource="@xml/automotive_app_desc" />
<service android:name=".MediaBrowserService">
    <intent-filter>
        <action android:name=
          "android.media.browse.MediaBrowserService" />
    </intent-filter>
</service>
```

Code 1: Android Auto service declaration in manifest file

This filtering identified 37 apps where developers explicitly declared intent to support Android Auto. We refer to this dataset as ***Corpus-F***. These apps represent developers' attempts to implement Android Auto support, making them ideal for studying compliance challenges.

Issue Collection. For the 37 apps in Corpus-F, we mined their GitHub/GitLab repositories to identify Android Auto-specific compliance issues. We performed keyword-based searches on issue titles and descriptions using the following terms: ("`Android Auto`", "`car screen`", "`car app`", "`automotive`", "`MediaBrowserService`", "`MediaSession`", "`voice command`", and "`Google Assistant`") followed by manual inspection to ensure comprehensive coverage.

Manual Filtering. Two authors with Android development experience independently reviewed all candidate issues to filter out false positives (e.g., feature requests unrelated to compliance, general Android bugs that coincidentally mention "Auto"). We retained only issues meeting three criteria: (1) describe failures specific to the Android Auto environment (not reproducible on phones), (2) relate to Android Auto APIs (`MediaBrowserService`, `MediaSessionCompat`, voice intents), and (3) involve mandatory platform requirements rather than optional features.

Of the 37 apps in Corpus-F, we found 98 Android Auto-specific issues in 14 apps that had public issue trackers with relevant reports.

2.2 Analysis Methodology.

Classification Process. The two authors independently classified each issue based on its manifestation UI (rendering failures), Media (playback control errors), Voice (command handling failures) and root cause following a systematic approach. Initially, each author reviewed all 98 issues and proposed categories based on issue descriptions, code changes in fixes, and affected Android Auto components. The authors then collaboratively refined these categories, merging overlapping ones and clarifying definitions. Next, both authors independently re-classified all issues using the refined definitions.

The issues with classification disagreements were resolved through discussion until consensus was reached.

Final Taxonomy. This systematic process yielded three primary violation categories based on the Android Auto component or the requirement violated:

- **T1 - Media Playback:** Violations related to `MediaSessionCompat` callbacks and playback control.
- **T2 - User Interface:** Violations related to `MediaBrowserService` callbacks and content hierarchy.
- **T3 - Voice Commands:** Violations related to the handling of voice action intent and Assistant integration.

These categories map directly to Android Auto's documented requirements [8] for media app compliance, suggesting that our taxonomy captures the platform's core compliance dimensions.

2.3 Results: Violation Categories

Our analysis identified 98 Android Auto issues across 14 repositories. Table 1 presents the complete taxonomy showing issue types, their root causes, and representative examples. Media playback violations dominate (59 issues, 60.2%), reflecting that most Android Auto apps are media players where playback control is fundamental functionality. UI violations account for 31 issues (31.6%), while voice command violations represent 8 issues (8.2%).

2.3.1 T1: Media Playback Violations (60.2%). Media playback violations involve incorrect or incomplete `MediaSessionCompat` callbacks implementation that control audio playback in response to user actions in the vehicle interface.

Platform-Specific Requirement for Media Playback: Android Auto requires apps to implement MediaSessionCompat.Callback methods (`onPlay()`, `onPause()`, `onStop()`, `onSkipToNext()`, `onSkipToPrevious()`, `onPlayFromMediaId()`) that are invoked by the vehicle system, not by user touches within the app, in response to dashboard button presses or steering wheel controls.

Table 1: Android Auto Issue Types and Root Causes.

Type	Root Cause	Issue Example	Issue Description
T1: Media Playback	Incomplete callback implementation	AntennaPod #2380	After completing a podcast episode, the app displays the episode list instead of showing information about the next podcast being played.
	State management errors	Vinyl Music Player #348	The app crashes when a song is liked using Android Auto.
T2: User Interface	Missing callbacks	Harmony-Music #111	No user interface displayed in Android Auto.
	Incorrect hierarchy structure	Podcini #57	The Android Auto UI has been broken for several releases.
T3: Voice Commands	Missing intent filter	Ultrasonic #827	App fails to support all required voice commands, such as not switching to the requested audio track when a voice command is issued.
	Unimplemented callback	Vinyl Music Player #376	Required voice commands not supported, such as failing to start the requested track after recognizing the command.

Root Causes. As shown in Table 1, media playback violations stem from two primary root causes. First, *incomplete callback implementation* occurs when developers implement basic playback callbacks (`onPlay()`, `onPause()`) but omit others (`onSkipToNext()`, `onSkipToPrevious()`, `onStop()`). This partial implementation works on phones where users interact with on-screen buttons, but fails in vehicles where all controls must work through callbacks. Second, *state management errors* occur when callbacks fail to properly update playback state (playing, paused, stopped) or metadata (current track, duration), causing the vehicle display to show incorrect information or become unresponsive.

Why This is Auto-Specific. In standard Android apps, playback controls are typically triggered by onClick listeners attached to UI buttons. In Android Auto, the vehicle system invokes callbacks even without the app's UI being visible, requiring a fundamentally different implementation pattern.

2.3.2 T2: User Interface Violations (31.6%). UI violations involve incorrect or missing implementation of `MediaBrowserService` callbacks that populate the vehicle's display with browsable content hierarchies.

Platform-Specific Requirement for User Interface: Android Auto requires apps to implement `MediaBrowserService` callbacks (`onGetRoot()`, `onLoadChildren()`) that return structured content hierarchies. The vehicle system invokes these callbacks to build the User Interface, apps cannot directly control when or how their content appears.

Root Causes. Table 1 identifies two root causes for UI violations. *Missing callbacks* occur when apps declare MediaBrowserService but fail to implement `onGetRoot()` or `onLoadChildren()`, resulting in blank screens. *Incorrect hierarchy structure* happens when apps implement callbacks but return improperly structured MediaItem objects (missing required metadata, incorrect flags, or invalid parent-child relationships), causing rendering failures or navigation errors.

Why This is Auto-Specific. Standard Android apps control UI rendering through Activities and direct view manipulation (`findViewById()` and `setContentView()`). Android Auto inverts this model: apps must describe their content structure through callbacks, and the vehicle system decides when and how to render it. This inverted control model is unique to Android Auto.

2.3.3 T3: Voice Command Violations (8.2%). Voice command violations involve missing or incorrect handling of voice action intents necessary for hands-free operation.

Platform-Specific Requirement to avoid Distraction: Android Auto requires apps to handle `MEDIA_PLAY_FROM_SEARCH` intents and implement `onPlayFromSearch()` callback to enable Google Assistant voice commands which is mandatory for safety compliance.

Root Causes. As detailed in Table 1, voice command violations have two primary causes. *Missing intent filters* occur when apps fail to declare the `MEDIA_PLAY_FROM_SEARCH` intent filter in their manifest, preventing Google Assistant from routing voice commands to the app. *Unimplemented callbacks* happen when apps declare the intent filter but don't implement `onPlayFromSearch()`, causing voice commands to be silently ignored.

Why This is Auto-Specific. Voice command integration is optional for standard Android apps but mandatory for Android Auto due to safety requirements: Drivers must be able to control apps without looking at or touching the screen.

2.4 Discussion and Implications

Answer to RQ1: Our study reveals that Android Auto compliance violations fall into three categories corresponding to the platform's core requirements: media playback control (60.2%), UI content provision (31.6%), and voice command handling (8.2%). These violations mostly stem from Android Auto's inverted control model, where the vehicle system, not the app, initiates callbacks to render UI and handle user interactions.

Comprehensiveness of Categories. Our three categories map directly to Android Auto's documented compliance requirements [8]: T1 corresponds to MediaSession callback requirements, T2 corresponds to MediaBrowser hierarchy requirements, and T3 corresponds to voice action requirements. This mapping suggests our taxonomy captures the platform's fundamental compliance dimensions. While additional edge cases may exist, these three categories represent the mandatory requirements all Android Auto media apps must satisfy.

Toward Automated Detection. Our analysis of 98 Android Auto violations reveals recurring patterns with similar fixes across apps. For example, the missing `onGetRoot()` and `onLoadChildren()` UI callbacks (Section 3) appears in multiple repositories with identical solutions. This recurrence, combined with evidence of developers referencing fixes from other apps, suggests that compliance knowledge can be generalized and automated. Table 1 shows that violations cluster around three core requirements with identifiable root causes. We systematically extracted compliance checking rules

from these violations and designed AutoComply, targeting Android Auto-specific patterns: missing MediaSession callbacks (T1), incorrect MediaBrowserService implementations (T2), and absent voice command handling (T3).

3 Motivating Example

To illustrate the importance of Android Auto compliance checking, we present a real-world case where a subtle implementation gap completely breaks the in-vehicle user experience. This example, based on developer reports and issue tracking systems, highlights the type of issue that existing static analysis tools fail to detect, but our AutoComply approach can identify.

Case Study: Missing Media Browser Interface. Consider a music streaming app with implementation shown in Listing 2 that works flawlessly on Android phones but fails to display any user interface when connected to Android Auto (similar to Fig 1). Users report that while the app appears in the Android Auto launcher, selecting it results in a blank screen with no visible playlists, albums, or songs. This creates a frustrating user experience where drivers are unable to access their music library through the vehicle's infotainment system.

The Problem Manifestation. When users connect their device to Android Auto and launch the music app, they encounter the following behavior: *(i)* The app icon is present in the car's infotainment system. *(ii)* The media browsing interface does not appear. *(iii)* Background audio playback works if started from the phone, but no in-car controls are available. *(iv)* Android Auto system logs show connection attempts but do not establish a media hierarchy.

This issue represents a complete failure of Android Auto integration despite the app working fine on the phone itself. From a safety perspective, this forces drivers to interact with their phone directly, exactly the behavior Android Auto was designed to prevent.

Root Cause Analysis. Our investigation reveals that the application implements a `MediaBrowserService` but provides incomplete callback implementations. Specifically, the developer implemented the service class and basic lifecycle methods but failed to properly implement the media browsing callbacks required for Android Auto's UI rendering.

The Correct Implementation. Listing 3 presents the corrected implementation that resolves the user interface issue. This version returns valid `BrowserRoot` objects with the appropriate media hierarchy identifiers and properly implements the asynchronous media loading pattern by always calling `result.sendResult()`. It also includes client validation for security and compliance, and provides a structured media hierarchy that Android Auto can use to generate and display the user interface as intended.

3.1 Why Existing Tools Cannot Detect This Issue

Standard static analysis tools fail to detect Android Auto compliance violations due to fundamental architectural mismatches:

The Core Problem: Traditional static analyzers like FlowDroid [4] model only *app-initiated* control flows, missing the *host-initiated* callback invocations that define Android Auto's execution model.

```
public class MusicService
    extends MediaBrowserServiceCompat {

    @Override
    public void onCreate() {
        super.onCreate();
        // Initializes MediaSession only
    }

    // CRITICAL: Missing onGetRoot() and onLoadChildren()
    // RESULT: Android Auto cannot build UI hierarchy
}
```

Code 2: Problematic implementation

```
public class MusicService
    extends MediaBrowserServiceCompat {

    @Override
    public void onCreate() {
        super.onCreate();
        // MediaSession initialization
    }

    // NEW: Host-invoked callbacks
    @Override
    public BrowserRoot onGetRoot(...) {
        if (isValidAutoClient(...)) {
            return new BrowserRoot(
                MEDIA_ROOT_ID, null);
        }
        return new BrowserRoot(
            EMPTY_ROOT_ID, null);
    }

    @Override
    public void onLoadChildren(...) {
        ...
        result.sendResult(mediaItems);
    }
}
```

Code 3: Corrected implementation

Figure 3: Comparison showing the missing callbacks (listing 2) that cause UI failures and their correct implementation (listing 3) that enables proper Android Auto rendering.

Missing Entry Points. FlowDroid [4] and similar tools construct ICFGs from standard Android entry points (Activity lifecycle methods). However, critical Android Auto callbacks like `onGetRoot()` and `onLoadChildren()` are invoked by the vehicle's infotainment system, not by the app or Android framework. These host-driven callbacks never appear as entry points in traditional analysis.

Incomplete Control Flow. Traditional ICFG analysis captures:

`onCreate()` → `bindService()` → *[analysis stops here]*

The vehicle system's subsequent invocations of `onGetRoot()` and `onLoadChildren()` occur outside this flow, making them invisible to analysis.

Superficial Lint Checks. Android Lint [16] includes only basic Auto checks (e.g., voice intent filters, service registration) but doesn't model the service-callback architecture required for proper content rendering.

Our Solution. AutoComply constructs a Car-Control Flow Graph (CCFG) that explicitly models host invocations through synthetic edges (Figure 2), capturing the complete execution path including external callback invocations.

3.2 Lessons Learned: Why Android Auto Requires Specialized Analysis

Challenge 1: Inverted Control Architecture. Standard Android apps control UI rendering through direct method calls (`setContentView()`,

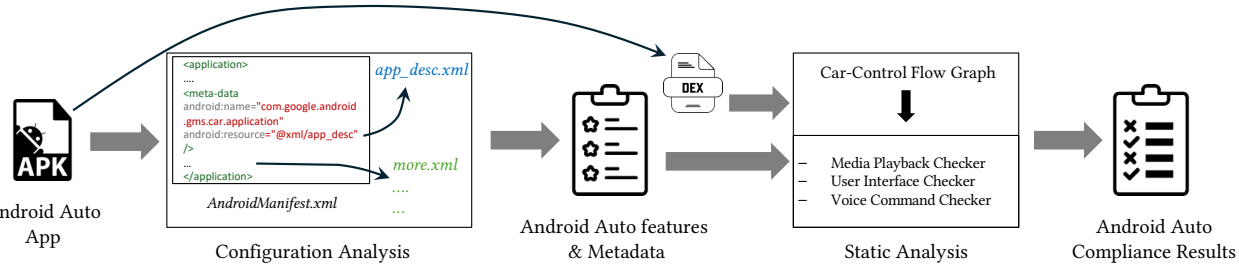


Figure 4: Overview of AutoComply: the process starts from configuration analysis to extract Android Auto features and metadata, followed by static analysis, and generating Android Auto compliance results.

`findViewById()`). Android Auto inverts this model: the vehicle system controls *when* and *how* to request content by invoking app callbacks. Developers must thus ensure their apps respond correctly to these external invocations by implementing auto specific callbacks.

Challenge 2: Safety-Critical Silent Failures. Missing callbacks produce blank screens with no error messages, exceptions, or crashes. Drivers cannot access app functionality and must resort to unsafe phone interactions while driving, precisely what Android Auto was designed to prevent.

Challenge 3: Testing Gap. These issues pass all conventional validation: compiling without warnings, satisfying Android Lint checks, and working on phones yet fail exclusively in vehicle environments, making them undetectable through standard testing practices.

Summary. As Figure 2 illustrates, traditional ICFG approaches fundamentally cannot capture host-driven control flows essential for Android Auto compliance. This motivates our specialized static analysis approach detailed in the following (section 4).

4 Static Analysis For Android Auto

To address the challenges developers face while making their apps Android Auto compliant, we introduce AutoComply, a novel static analysis framework designed to systematically detect and analyze issues in Android Auto apps. AutoComply focuses on the three primary issue types identified in our study: *T1: Media Playback Issues*, *T2: User Interface Issues*, and *T3: Voice Command Issues*. For each issue type, AutoComply employs dedicated checkers – *C1, C2, and C3* – to detect specific compliance violations. In the following, we describe the architecture of AutoComply, the construction of the CCFG, and the compliance checks.

4.1 Overview of AutoComply

As illustrated in Figure 4, the analysis proceeds in three stages:

Configuration Analysis. AutoComply begins by parsing configuration artifacts (e.g., `AndroidManifest.xml`) to extract Android Auto-specific metadata, such as `<meta-data>` entries, service declarations, and intent filters. This step identifies candidate components (e.g., `MediaBrowserService`, `MediaSessionCompat`) that indicate support for Android Auto.

Car-Control Flow Graph Construction. Based on the extracted metadata, the framework extends a traditional ICFG to include *host-driven* control flows. These represent interactions where the vehicle's infotainment system or Assistant invokes app callbacks (e.g., `onGetRoot()`, `onPlayFromSearch()`) directly. The resulting CCFG models both app-initiated and host-initiated execution, enabling whole-app compliance reasoning.

Compliance Checking. Specialized checkers traverse the CCFG to identify violations of compliance categories identified in our formative study: *(C1)* media playback, *(C2)* User Interface, and *(C3)* voice-command handling. Each checker verifies the presence, connectivity, and semantic correctness of the relevant callbacks.

This analysis pipeline connects configuration data with code-level control flow, producing actionable reports to help developers address compliance issues.

4.2 Car-Control Flow Graph Construction

To enable static reasoning under Android Auto's host-invoked execution model, *Car-Control Flow Graph (CCFG)* augments the app's control-flow graph with host-driven entry points.

Given an app A with a traditional ICFG $G_{\text{base}} = (V_{\text{base}}, E_{\text{base}})$, AutoComply constructs: $G_{\text{car}} = (V_{\text{base}} \cup V_{\text{host}},\ E_{\text{base}} \cup E_{\text{host}})$,

where each node $v_h \in V_{\text{host}}$ represents a callback invoked by the host (e.g., `onGetRoot()`, `onLoadChildren()`, `onPlayFromSearch()`). Each edge $(v_i, v_h) \in E_{\text{host}}$ models a host-triggered invocation, derived by resolving manifest declarations (e.g., `<meta-data>`, `<service>`) and mapping them to their implementing service classes.

Figure 2 illustrates this augmentation. New edges (shown in green) connect system-invoked methods to their corresponding app callbacks, capturing control transfers that traditional static analyses miss. By modeling these host-initiated paths, AutoComply enables precise reasoning about execution flows that underlie UI rendering, media playback, and voice-command support.

Algorithm 1 formalizes this process: Lines 1–9 of Algorithm 1 capture this construction process. ExtractMetadata reads metadata *autoMetadata*, ConstructBaseICFG builds G_{base}, InitializeCCFG prepares the structure, and GetCallbacks enumerates and links Android Auto callbacks to their respective components. The following specialized checkers work on lines 10–14 of Algorithm 1 and check whether these callbacks are implemented according to Android Auto rules, such as returning valid browsing hierarchies and using the correct flags.

4.3 Compliance Checkers

AutoComply employs three specialized checkers: *C1* – Media Playback Checker, *C2* – User Interface Checker, and *C3* – Voice Command Checker, to address the specific issue types identified in our study (Sec. 2).

Algorithm 1: Android Auto Compliance Analysis

Input: An APK file apk
Output: Detected compliance issues

$autoMetadata \leftarrow$ ExtractMetadata($apk.manifest$)
$G_{base} \leftarrow$ ConstructBaseICFG(apk)
$ccfg \leftarrow$ InitializeCCFG(apk)
$requiredCallbacks \leftarrow$ InitializeCallbacks()
foreach $autoComponent$ ***in*** $autoMetadata.components$ **do**
  $callbacks \leftarrow$ GetCallbacks($autoComponent$)
  AddCallbackNodes($ccfg, callbacks$)
  ConnectCallbackEdges($ccfg, callbacks$)
end
foreach cb ***in*** $requiredCallbacks$ **do**
  if $\neg$*IsCompliant*($ccfg, autoComponent, cb$) **then**
    report auto compliance issue
  end
end

C1 – Media Playback Checker. Android Auto requires apps to interact with the vehicle's infotainment system through a specific set of callbacks that are invoked by the platform at runtime. These callbacks are not triggered by user actions within the app itself, but rather by the Android Auto runtime in response to driver interactions with the car's interface (screen or hardware buttons) or through voice commands. As a result, developers must implement methods such as `onPlay()`, `onPause()`, `onStop()`, `onSkipToNext()`, `onSkipToPrevious()`, and `onPlayFromMediaId()` to handle media playback requests generated externally by the vehicle system.

To assess compliance, the checker begins by extracting these media-related callbacks as nodes in the CCFG. It verifies the presence of all required methods in the `MediaSessionCompat.Callback` implementation in the app and confirms that each callback is properly connected within the control-flow graph. Through dataflow analysis, the checker examines whether these callbacks invoke the appropriate playback or resource management logic, validating that media actions requested by the platform are handled correctly. The analysis also evaluates whether the app manages resources such as large media files and buffers effectively within each callback, and whether any error conditions or edge cases are addressed to prevent failures or degraded performance during playback. Any missing, incomplete, or incorrectly implemented callbacks, as well as resource management or playback handling issues, are reported as compliance violations. All findings are mapped to specific CCFG nodes to provide developers with clear guidance for correcting media playback problems and meeting Android Auto compliance requirements.

C2 – User Interface Checker. In Android Auto, vehicle's infotainment relies on callbacks such as `onGetRoot()` and `onLoadChildren()` to dynamically populate and display media or navigation hierarchies for the driver. The checker begins by identifying the relevant `MediaBrowserServiceCompat` components in the app and locating the corresponding nodes for `onGetRoot()` and `onLoadChildren()` in the CCFG. It then checks that both methods are present with the correct signatures and analyzes their control-flow to verify that all execution paths return valid `BrowserRoot` objects (in `onGetRoot()`) and that `sendResult()` is called as required in `onLoadChildren()`.

The checker also validates that the app provides a non-empty and properly structured hierarchy of media or navigation items, which is essential for Android Auto to generate a usable user interface on the dashboard. In addition, the analysis reviews client validation logic and error handling to identify cases that could result in blank screens, empty lists, or unresponsive menus for the driver. Any deficiencies, such as missing callbacks, incorrect method signatures, incomplete media hierarchies, or weak error handling, are flagged as user interface compliance violations.

C3: Voice Command Checker. For safety and distraction-free driving, Android Auto requires apps to fully support voice interactions so that drivers can control playback and navigation without taking their hands off the wheel. The Voice Command Compliance Checker uses the CCFG to evaluate whether an app implements the platform's requirements for voice command support. In the Android Auto environment, the system invokes `onPlayFromSearch()` callback, in response to driver voice commands or requests through the Google Assistant. Unlike callbacks in standard Android apps, these methods are triggered externally by the platform rather than through direct user interaction within the app interface.

To evaluate compliance, the checker identifies all Assistant-driven callbacks in the CCFG, focusing on `onPlayFromSearch()` within the app's `MediaSessionCompat.Callback` implementation. It verifies the presence and correct signature of these methods, and applies control-flow analysis to ensure that voice command inputs are processed correctly. The checker also examines whether command parameters are validated, playback state is updated appropriately, and errors or unrecognized requests are handled gracefully to avoid silent failures. Any missing, incomplete, or incorrectly implemented voice command handlers, as well as improper handling of command inputs or error conditions, are reported as voice compliance violations.

5 Evaluation

This section evaluates AutoComply on real-world Android Auto apps, demonstrating its effectiveness in detecting compliance violations, its advantages over existing tools, and its practical value to developers. We address the following research questions:

- **RQ2 (Effectiveness):** Can AutoComply effectively and precisely detect Android Auto compliance issues in real-world apps?
- **RQ3 (Comparison):** How does AutoComply compare with existing tools in coverage and accuracy?
- **RQ4 (Usefulness):** Do developers acknowledge and address issues detected by AutoComply?

5.1 Methodology

To evaluate compliance violations in Android Auto apps, we compiled a dataset of open-source apps from GitHub [13] by performing a code search for repositories with Android Auto related declarations in their `AndroidManifest.xml`. The keywords we used are `com.google.android.gms.car.application` and `android.media.browse.MediaBrowserService`. We collected apps that *(i)* have commits after January 2024 (i.e., actively-maintained), *(ii)* have at least 30 stars and are not forks, *(iii)* have a public issue

Table 2: Detailed static analysis results when Android Lint and AutoComply applied on selected Android Auto apps. Stars: Number of GitHub stars; Downloads: Downloads on Google Play; KLOC: Source code size (thousands of lines); Commit: Latest commit hash; TP: True Positives; FP: False Positives; Time(s): Analysis time in seconds.

						Android Lint			AutoComply		
#	Repository	Stars	Downloads	KLOC	Commit	TP	FP	Time	TP	FP	Time
1	anandnet/Harmony-Music [1]	1.4K	–	35	ff91d19	0	0	1	3	0	1
2	antoinepirlot/Satunes [2]	38	–	138.8	24e3933	0	0	1	2	0	18
3	DJDoubleD/refreezer [9]	433	49K	34.7	aecf242	0	0	1	1	0	4
4	gokadzev/Musify [15]	2.4K	467K	9.5	f524674	0	0	1	3	0	2
5	jellyfin/jellyfin-android [20]	1.7K	1M+	158.9	7886df9	0	0	140	2	0	7
6	KRTirtho/spotube [21]	33.9K	–	68.2	8c1337d	0	0	1	3	0	2
7	LISTEN-moe/android-app [25]	258	–	90.4	4194aeb	0	0	30	2	0	3
8	namidaco/namida [32]	2.9K	289K	92.9	675dc79	0	0	1	2	0	3
9	nextcloud/news-android [33]	705	5K+	223.5	9636d61	0	0	18	1	0	5
10	nt4f04uNd/sweyer [34]	209	–	27.5	e5abe5a	0	0	1	3	0	2
11	OxygenCobalt/Auxio [35]	2.3K	–	47.9	fce77ec	1	0	198	1	0	6
12	quran/quran_android [36]	2.1K	50M+	58.2	f5bd3dc	0	0	138	1	0	6
13	Simple-Music-Player [40]	1.3K	–	215.3	498086a	0	0	64	1	0	8
14	sosauce/CuteMusic [41]	363	4K	30.4	97bb20b	1	0	119	1	0	2
15	timusus/Shuttle2 [42]	208	–	295.5	6fd520f	0	0	315	1	0	21
Total Issues / Geomean Time (s)						**2**	**0**	**11.1**	**27**	**0**	**4.2**

tracking system (for submitting issue reports). Finally, we manually filtered the results for duplicates, library repositories, and plugins.

This dataset, *Corpus-G*, consists of 31 Android Auto apps with a total of 6.5 million lines of code. The availability of source code enables a direct comparison between AutoComply and Android Lint [16], facilitating a comprehensive assessment of their detection capabilities. The collected apps differ in functionality, size, and complexity, providing a well-rounded sample for analyzing compliance challenges in Android Auto development.

Android Lint [16] is a static analysis tool provided by Google to detect potential issues in Android apps. It includes basic compliance checks for Android Auto, primarily focusing on media playback compliance. In our evaluation, we use Android Lint as a baseline to compare its issue detection capabilities against AutoComply, highlighting differences in coverage and accuracy.

Our experiments were conducted on an Apple MacBook Pro equipped with Intel Core i7 2.7 GHz Quad-Core processors and 16GB of memory, reflecting a typical system configuration accessible to most app developers. For testing on a physical device, we utilized a Google Pixel 7a running Android 14.0.

5.2 RQ2: Effectiveness of AutoComply

To evaluate the applicability of AutoComply, we applied it to Corpus-G, a dataset of 31 open-source Android Auto apps spanning diverse categories. The objective was to assess whether AutoComply could effectively detect compliance issues across various real-world apps and remain computationally efficient for large scale analysis. Our analysis identified 27 compliance issues across 15 apps, spanning media playback, UI constraints, and voice command handling. These findings demonstrate that Android Auto compliance challenges are common even in actively maintained apps. The geometric mean analysis time per app was 4.2 seconds, showing that AutoComply can efficiently analyze apps of varying complexity without significant overhead.

The evaluation followed a two-step process. First, we performed static analysis on the selected apps to identify potential compliance violations. Next, we tested these apps in a real Vehicle to verify the detected issues in a real execution environment.

All identified issues were manually validated, confirming their impact on app behavior. As discussed later in this paper, the verified issues were also reported to developers, reinforcing the practical relevance of AutoComply. This evaluation examines the tool's ability to detect Auto compliance violations accurately while minimizing incorrect detections. The results are summarized in Table 2, under the AutoComply columns.

Issue Detection Accuracy. Across the evaluated apps, AutoComply correctly identified 27 compliance issues without reporting any false positives or missing any expected violations. The absence of false negatives shows that the tool successfully captures all relevant compliance issues, while the lack of false positives indicates that its detections are highly reliable. These results highlight AutoComply's strong detection capabilities, providing developers with precise and actionable findings.

Discussion. As mentioned previously, cross-platform apps may implement Android Auto functionality differently, requiring additional support in AutoComply to handle such variations. Similarly, apps incorporating C/C++ may introduce compliance-related logic outside the primary Java/Kotlin codebase, necessitating further enhancements. While AutoComply performs exceptionally well for Java/Kotlin-based Android Auto apps, its applicability to multi-language projects requires further evaluation. Many modern Android Auto apps integrate JavaScript (e.g., React-Native), Dart (i.e. Flutter) for functionalities such as UI rendering or media playback. Since AutoComply primarily analyzes Java/Kotlin code, it may not detect compliance-related logic implemented in other languages, potentially leading to false positives in hybrid apps.

Answer to RQ2: AutoComply achieves high accuracy in detecting Android Auto compliance issues, correctly identifying 27 issues with no false positives or false negatives.

These results demonstrate that AutoComply provides reliable checks on real-world Android Auto apps.

5.3 RQ3: Comparison with Existing Tools

To evaluate AutoComply against existing Android Auto analysis tools, we compare it with Android Lint [16], which, to the best of our knowledge, is the only tool that includes some built-in checks for Android Auto compliance violations, specifically for verifying that the `MediaBrowserService` has the necessary intent filters declared in the app's manifest. Due to its native support for Android Auto-related analysis, we use Android Lint as the baseline for comparison.

Issue Coverage and Accuracy. The results in Table 2 demonstrate that AutoComply provides significantly broader issue coverage than Android Lint. While Android Lint detected only 2 compliance issues, AutoComply identified 27 issues across the same set of 15 apps - a 13X increase. This suggests that Android Lint's built-in compliance checks, which focus only on voice search issues, may overlook many real-world problems. In contrast, AutoComply is specifically designed to capture Android Auto-related constraints, including UI constraints, media playback and voice command handling, which are critical for compliance. In addition, AutoComply was able to detect both issues detected by Android Lint.

Both tools reported zero false positives, meaning that every detected issue was a valid compliance violation. However, AutoComply's ability to detect a broader range of issues demonstrates its higher sensitivity to real compliance problems that Android Lint does not identify.

Performance Efficiency. AutoComply achieves significantly better performance, analyzing apps in a geometric mean of 4.2 seconds, compared to 11.1 seconds for Android Lint- a more than 2X speed improvement. This efficiency makes AutoComply more practical for real-world use, allowing developers to detect issues quickly with minimal overhead.

This performance gain is due to fundamental differences in how the tools operate. Android Lint analyzes source code and requires processing to multiple source files, which increases processing time. In contrast, AutoComply operates directly on APK files and leverages the CCFG to model app execution more efficiently. This enables faster and more precise detection of compliance issues, making AutoComply both more effective and computationally efficient.

> **Answer to RQ3:** AutoComply significantly outperforms Android Lint by detecting 27 issues compared to only 2, while both tools report zero false positives. Additionally, AutoComply runs more than 2X faster, making it a more comprehensive and efficient solution for Android Auto compliance analysis.

5.4 RQ4: Usefulness of AutoComply

To evaluate whether developers find AutoComply's detected violations actionable, we reported all 27 compliance issues to their respective app developers through their GitHub issue trackers. Each report followed a structured format including: (1) clear issue description with observable symptoms, (2) root cause analysis identifying the missing or misconfigured Android Auto API, (3) links to official Android Auto documentation, and (4) references to successful implementations in similar apps. To avoid overwhelming maintainers, we consolidated all detected issues for each app into a single, well-documented bug report.

Discovered Issues Overview. AutoComply detected 27 compliance violations across 15 apps, distributed as follows: 10 UI violations (T2), 7 media playback violations (T1), and 10 voice command violations (T3). The detected issues span all three violation categories from our taxonomy, validating that AutoComply provides comprehensive coverage of Android Auto compliance requirements. UI violations primarily involved missing `onGetRoot()` or `onLoadChildren()` callbacks that prevented content from displaying on vehicle screens. Media playback violations included incomplete callback implementations (e.g., missing `onStop()`) that caused unexpected behavior during playback control. Voice command violations consisted of missing intent filters or unimplemented `onPlayFromSearch()` callbacks that prevented Google Assistant integration. Notably, several apps had multiple violation types, for instance, some apps had both UI rendering issues and voice command gaps, indicating that Android Auto compliance requires attention across multiple platform requirements simultaneously. All detected violations matched patterns identified in our formative study, demonstrating that AutoComply successfully generalizes compliance checking rules beyond the training dataset.

Developer Response and Engagement. Of the 27 reported issues, developers have responded to 14 so far. Eight issues have been fixed and merged into production, while six acknowledged issues are scheduled for future releases.

Developer responses fell into three patterns. First, some developers responded enthusiastically with immediate fixes, releasing updated versions and requesting verification that issues were resolved. These developers treated the reports as high-priority user experience improvements. Second, other developers acknowledged the issues positively but indicated resource constraints, expressing appreciation for the detailed reports with documentation references and noting plans to address the issues when time permits. One developer described Android Auto support as requiring 'minimal tinkering' and committed to keeping it on the development radar. Third, some developers sought community assistance due to time constraints, with one requesting pull request contributions, a response pattern enabled by our detailed reports with working examples.

Notably, one developer acknowledged the potential user experience improvement but expressed broader concerns about framework dependencies that influenced their development priorities. Despite varying circumstances, all fixed issues were addressed within one to four weeks, demonstrating that AutoComply identifies actionable problems developers find valuable. The response variation reflects typical open-source dynamics where maintainer availability and project priorities influence fix timelines rather than issue validity.

Impact on App Quality. For the 8 fixed issues, we verified that fixes were correctly implemented and resolved the compliance violations. We re-ran AutoComply on updated versions and confirmed zero violations in the previously flagged categories. These fixes directly improved user experience by enabling previously broken functionality, apps that showed blank screens now properly display content hierarchies, media playback controls now function

correctly across all callbacks, and voice commands now work as intended. The fixes demonstrate that AutoComply detects genuine compliance violations that, once addressed, measurably improve Android Auto app quality and safety.

> **Answer to RQ4:** AutoComply effectively identifies actionable compliance violations that developers find valuable. All fixed issues were verified to resolve the detected violations. The positive reception, rapid fix adoption, and measurable quality improvements demonstrate that AutoComply provides practical value to developers.

6 Threats to Validity

External Validity. AutoComply was primarily evaluated on apps written in Java and Kotlin, limiting its applicability to cross-platform apps using C#, Xamarin, or native C/C++. While many of these platforms do not currently support Android Auto, their exclusion may affect the generalizability of our results.

Construct Validity. Our evaluation focuses on specific Android Auto compliance issues but does not cover runtime performance or user interaction patterns, meaning certain issues may go undetected. Code obfuscation also poses a challenge, as it modifies class names and control flows, potentially impacting AutoComply' accuracy and leading to false positives.

Conclusion Validity. Analysis time varies with app complexity, with more intricate apps requiring additional processing time. While this does not affect detection accuracy, longer runtimes could impact developer workflows. Additionally, although we include widely used apps in our dataset, the corpus may not fully represent the entire Android Auto ecosystem, which could influence the broader applicability of our conclusions.

7 Related Work

Mobile app reliability and compatibility issues have been extensively studied in the Android ecosystem [6, 19, 22, 26, 38, 43, 44]. Wei et al. [44] characterized fragmentation-induced compatibility issues and proposed FicFinder for static detection. Their follow-up work, Pivot [45], identifies compatibility problems across Android versions and devices. Huang et al. [19] developed CIDER to detect callback compatibility issues from API evolution, while Li et al. [22] created CiD to identify incompatibilities from API changes. These works focus on device fragmentation and OS version compatibility within standard Android, whereas our work addresses platform-specific compliance for Android Auto's unique service-based architecture where the vehicle system invokes app callbacks externally.

Android Auto-specific research remains limited. Zhang et al. [47] developed techniques to test message synchronization and UI consistency between phones and vehicle dashboards, focusing primarily on messaging functionality. While valuable, their work does not address broader compliance challenges including service-based architecture requirements, media playback callbacks, or voice command integration that affect most Android Auto apps. Mandal et al. [29, 30] explored security vulnerabilities in Android Auto apps, focusing on privacy leaks and malicious behaviors. Their work assumes functional compliance and targets security issues, whereas we address the fundamental implementation gaps preventing apps from working correctly in Android Auto environments.

Static analysis tools like FlowDroid [4] provide precise taint analysis for Android apps through ICFG construction. However, as we demonstrate in Section 3, traditional ICFGs fundamentally cannot capture host-driven callbacks essential for Android Auto compliance because they only model app-initiated flows. Android Lint [16] includes basic Android Auto checks, primarily verifying manifest declarations, but our evaluation (Section 5) shows it detects only 2 issues compared to AutoComply's 27. Lint's rule-based approach lacks the deep control-flow analysis necessary to detect missing callback implementations or media playback handlers.

Other research has addressed race conditions and energy bugs [17, 18, 28], network and GPS issues [23], app quality [3, 5, 24, 27, 31], configuration changes [11, 12, 37], and memory leaks [46]. While these contributions improve mobile app quality, they target general Android problems rather than platform-specific compliance requirements unique to Android Auto.

Our work provides the first comprehensive study of Android Auto compliance through analysis of 98 real-world issues, introduces the Car-Control Flow Graph (CCFG) to model host-invoked callbacks invisible to traditional analysis, and demonstrates practical impact with 27 detected violations and 8 developer-confirmed fixes. Unlike existing tools that focus on general Android compatibility or security, AutoComply systematically detects compliance violations specific to Android Auto's inverted control architecture.

8 Conclusion

Despite the vast number of Android apps available, numbering in the millions, the number of Android Auto-compliant apps remains remarkably small. This disparity highlights the significant challenges that developers face in adapting their apps for the automotive platform. In this paper, we conducted a comprehensive investigation into these challenges. Our study revealed that UI issues, media playback errors, and voice command integration errors are often caused by incompatibility, and developers are left to carry out manual issue detection. To address these challenges, we introduced AutoComply, a novel static analysis designed to detect compliance issues in Android Auto apps. AutoComply applies static analysis techniques to identify potential problems, providing developers with actionable insights to enhance the reliability of their apps. Our evaluation of AutoComply on 31 Android apps demonstrated its effectiveness, identifying 27 compliance issues. This work contributes to the limited body of research on Android Auto development by providing a practical solution for detecting compliance issues early in the development process. Moving forward, we aim to extend the capabilities of AutoComply to cover additional features and add support for dynamic analysis to detect runtime issues.

Acknowledgments

This material is based upon the work supported in part by the NSF under Grant No. 2449694 and the Louisiana Board of Regents. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or the Louisiana Board of Regents.

References

[1] anandnet. 2023. A cross platform app for streaming music. https://github.com/anandnet/Harmony-Music/. Accessed: 2024-08-14.

[2] Antoinepirlot. 2024. Modern MP3 Player to listen to your local music files on Android Lollipop 5.1.1+ & compatible with Android Auto. https://github.com/antoinepirlot/Satunes. Accessed: 2024-08-14.

[3] Niaz Arijo, Reiko Heckel, Mirco Tribastone, and Stephen Gilmore. 2011. Modular performance modelling for mobile applications. In *ACM SIGSOFT Software Engineering Notes*, Vol. 36. ACM, 329–334.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[5] Luca Berardinelli, Vittorio Cortellessa, and Antinisca Di Marco. 2010. Performance modeling and analysis of context-aware mobile software systems. *Fundamental Approaches to Software Engineering* (2010), 353–367.

[6] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 216–227.

[7] L. Ceci. 2023. Number of apps available in leading app store. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed: 2023-11-06.

[8] Android Developers. 2015. Android Auto. https://www.android.com/auto/. Accessed: 2024-09-03.

[9] DJDoubleD. [n. d.]. An alternative Deezer music streaming & downloading client, based on Freezer. https://github.com/DJDoubleD/refreezer. Accessed: 2024-08-14.

[10] F-Droid. 2010. F-Droid. https://f-droid.org/en/packages/. Accessed: 2024.

[11] Umar Farooq and Zhijia Zhao. 2018. RuntimeDroid: Restarting-Free Runtime Change Handling for Android Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (Munich, Germany) *(MobiSys '18)*. Association for Computing Machinery, New York, NY, USA, 110–122. https://doi.org/10.1145/3210240.3210327

[12] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. 2020. LiveDroid: Identifying and Preserving Mobile App State in Volatile Runtime Environments. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 160 (nov 2020), 30 pages. https://doi.org/10.1145/3428228

[13] Github. 2016. https://github.com. Accessed: 2024.

[14] Github. 2018. https://gitlab.com. Accessed: 2024-09-03.

[15] gokadzev. [n. d.]. Unlock the full potential of music: Stream effortlessly with one app! https://github.com/gokadzev/Musify. Accessed: 2024-08-14.

[16] Google. 2025. Android Lint Rules. https://googlesamples.github.io/android-custom-lint-rules/. Accessed: 2025-02-10.

[17] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. 2014. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices* 49, 6 (2014), 326–336.

[18] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, 77–83.

[19] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for android applications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 532–542.

[20] Jellyfin. 2020. Android Client for Jellyfin. https://github.com/jellyfin/jellyfin-android/. Accessed: 2024-08-14.

[21] KRTirtho. 2021. An open source cross-platform Spotify client. https://github.com/KRTirtho/spotube. Accessed: 2024-08-14.

[22] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.

[23] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. 2014. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In *Proceedings of the 20th annual international conference on Mobile computing and networking*. ACM, 519–530.

[24] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 445–456.

[25] Listen-moe. 2021. Official LISTEN.moe Android app. https://github.com/LISTEN-moe/android-app. Accessed: 2024-08-14.

[26] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. 2021. Identifying and characterizing silently-evolved methods in the android API. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 308–317.

[27] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1013–1024.

[28] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for android applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 316–325.

[29] Amit Kr Mandal, Agostino Cortesi, Pietro Ferrara, Federica Panarotto, and Fausto Spoto. 2018. Vulnerability analysis of android auto infotainment apps. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 183–190.

[30] Amit Kr Mandal, Federica Panarotto, Agostino Cortesi, Pietro Ferrara, and Fausto Spoto. 2019. Static analysis of Android Auto infotainment and on-board diagnostics II apps. *Software: Practice and Experience* 49, 7 (2019), 1131–1161.

[31] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software testing of mobile applications: Challenges and future research directions. In *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, 29–35.

[32] Namidaco. [n. d.]. A Beautiful and Feature-rich Music & Video Player with Youtube Support, Built in Flutter. https://github.com/namidaco/namida. Accessed: 2024-08-14.

[33] Nextcloud. 2013. Android client for the Nextcloud news/feed reader app. https://github.com/nextcloud/news-android. Accessed: 2024-08-14.

[34] nt4f04uNd. [n. d.]. Music player built with Flutter. https://github.com/nt4f04uNd/sweyer. Accessed: 2024-08-14.

[35] OxygenCobalt. 2020. A simple, rational music player for android. https://github.com/OxygenCobalt/Auxio. Accessed: 2024-08-14.

[36] Quran. [n. d.]. A quran reading application for android. https://github.com/quran/quran_android. Accessed: 2024-08-14.

[37] Sydur Rahaman, Umar Farooq, Iulian Neamtiu, and Zhijia Zhao. 2023. Detecting Potential User-data Save & Export Losses due to Android App Termination. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*. 152–162. https://doi.org/10.1109/AST58925.2023.00019

[38] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. 2019. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 288–298.

[39] Ben Schoon. 2024. Over 200 million cars have Android Auto, a decade after its debut. https://9to5google.com/2024/05/16/android-auto-number-of-cars-2024/. Accessed: 2024-09-03.

[40] SimpleMobileTools. [n. d.]. A clean music player with a customizable widget, stylish interface and no ads. https://github.com/SimpleMobileTools/Simple-Music-Player. Accessed: 2024-08-14.

[41] Sosauce. [n. d.]. CuteMusic is a simple,lightweight and open-source offline music player app for Android. https://github.com/sosauce/CuteMusic. Accessed: 2024-08-14.

[42] Timusus. [n. d.]. Shuttle Music Player 2.0. https://github.com/timusus/Shuttle2. Accessed: 2024-08-14.

[43] Haoyu Wang, Hongxuan Liu, Xusheng Xiao, Guozhu Meng, and Yao Guo. 2019. Characterizing android app signing issues. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 280–292.

[44] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 226–237.

[45] Lili Wei, Yepang Liu, Shing-Chi Cheung, Huaxun Huang, Xuan Lu, and Xuanzhe Liu. 2018. Understanding and detecting fragmentation-induced compatibility issues for android apps. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1176–1199.

[46] Dacong Yan, Shengqian Yang, and Atanas Rountev. 2013. Systematic testing for resource leaks in Android applications. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 411–420.

[47] Yu Zhang, Xi Deng, Jun Yan, Hang Su, and Hongyu Gao. 2019. Testing the Message Flow of Android Auto Apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 559–563. https://doi.org/10.1109/SANER.2019.8667973