Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

# Static Analysis of Android Auto Infotainment and ODB-II Apps

Amit Kr Mandal[1,2] | Federica Panarotto[3] | Agostino Cortesi[1] | Pietro Ferrara[4] | Fausto Spoto[3]

[1]Università Ca' Foscari Venezia, Italy
[2]BML Munjal University, Haryana, India
[3]University of Verona, Italy
[4]JuliaSoft Srl, Verona, Italy

**Correspondence**
Agostino Cortesi, DAIS, Università Ca' Foscari Venezia, Venice, Italy. Email: cortesi@unive.it

**Abstract**

Smartphone and automotive technologies are rapidly converging, letting drivers enjoy communication and infotainment facilities and monitor in-vehicle functionalities, via On Board Diagnostics (OBD) technology. Among the various automotive apps available in playstores, Android Auto infotainment and OBD-II apps are widely used and are the most popular choice for smartphone to car interaction. Automotive apps have the potential of turning cars into *smartphones on wheels*, but can also be the gateway of attacks. This paper defines a static analysis that identifies potential security risks in Android infotainment and OBD-II apps. It identifies a set of potential security threats and presents an actual static analyzer for such apps. It has been applied to most of the highly rated infotainment apps available in the Google Play store, as well as on the available open-source OBD-II apps, against a set of possible exposure scenarios. Results show that almost 60% of such apps are potentially vulnerable and that 25% pose security threats related to the execution of JavaScript. The analysis of the OBD-II apps shows possibilities of severe Controller Area Network (CAN) injections and privacy violations, because of leaks of sensitive information.

**KEYWORDS:**
Android Auto Security, OBD-II App Security, In-vehicle Infotainment System, Abstract Interpretation, Static Analysis

## 1 | INTRODUCTION

The average car or truck driver is nowadays an avid smartphone user. Recent studies by Zendrive on 3-million drivers show that they use their phones during 88 out of 100 trips with an average use of 3.5 minutes per hour, for various purposes[1]. To address this market scenario, car manufacturers started to offer apps to run on car's infotainment system for a more direct interaction with the car and its external environment. Hence, modern infotainment systems have evolved from the simple control of the music stereo or navigation system to being the hub of many vehicle functions such as telephone handling, data communication, vehicle setup and climate control. Infotainment systems are vendor-specific: every car manufacturer and every Electronic Control Unit (ECU) producer comes with a proprietary proposal. To ease up the technology landscape and reduce vendor locking, mobile software companies such as Google and Apple provide generic infotainment systems, so that many car manufacturers support Apple CarPlay[2] and Android Auto[3] now. Both are fairly similar and pump a small portion of the mobile phone's experience into a car's built-in infotainment system, letting the driver access some of the smartphone's functionalities. However, its two billion mobile users base sets Android as the most popular choice for automotive infotainment.

Infotainment and on-board diagnostics apps are the most popular in appstores for cars. They build on top of the Android Auto framework, that offers a lightweight informative interface similar to Google Now. It uses Google's unified design language and provides a card-based user interface. Many apps have been published on the Google Play store[4], with the necessary driver-safety measures in place. These apps facilitate the interaction with the multiple devices connected to the car via Android Auto. OBD-II (On-Board Diagnostics II) apps help in monitoring the car's health. In general, they use a OBD-II device plugged into the car's diagnostic port and communicate with the smartphone, for instance via Bluetooth. Although infotainment and OBD-II apps were intended to make driver's life easier, they also expose automobiles to potential attacks[5,6] that exploit internet or Bluetooth connections[7,8,9]. If adversaries gain access to the infotainment system, they can then play with safety-critical functions: they can alter the vehicle's electronic ID, jam with the radio-based and navigation system, spoof sensor data, interfere with control units, master data and firmware/software, just to name a few examples. Beside such attack scenarios from the outside world, infotainment apps can be very harmful if they distract the driver. Thus, infotainment and OBD-II apps security has become essential, especially in today's socio-technical landscape, where 70% of drivers engage in infotainment activities while driving[10].

Various techniques to mitigate the security risks of the infotainment and OBD systems have been proposed[11,12,13]. However the security of the modern automotive infotainment systems can be still greatly compromised by interfering with Bluetooth, Wi-Fi and telematics signals[8,14]. On the other side, connecting OBD devices directly to the vehicle is potentially dangerous, because the CAN (Controller Area Network), does not offer by design any protection against manipulation[5,6]. Furthermore, it is also possible to inject crafted CAN messages[15] via feasible attack surfaces in state-of-the-art bus systems to perform malicious activities[16,9]. The majority of the current scientific approaches primarily discussed the possible vulnerabilities, without any real solution that mitigates or at least locates the issues.

This article[1] introduces a static analysis based on abstract interpretation[19,20], that discovers potential software vulnerabilities in automotive apps. Many static analyzers for Java source code exist already and find bugs or inefficiencies. However, most of them perform mostly syntactical and unsound checks only (Checkstyle[2], Coverity[3], FindBugs[4], PMD[5]) or use theorem proving, with simplifying hypotheses. Unfortunately, these tools do not support features specific to Android, such as component interaction and XML inflation, that affect the construction of the control flow graph of an Android app. In contrast to that, QARK[21] is a comprehensive static analyzer for Android apps. It looks for a wide range of standard smartphone vulnerabilities, such as the use of WebViews, data broadcast and cryptography. However, QARK does not provide a solution tailored for Android automotive apps. Furthermore, it is not yet stable, as it crashed many times during our experiments. Instead, the Julia static analyzer[22] performs a semantic based analysis, which justifies our choice to develop our analysis on top of it. The overall architecture of our system is as follows. First, the apps are reverse-engineered through `dex2jar`[23], to extract their Java bytecode, and `apktool`[24], to extract their Android manifest. The manifest is then used to determine the entry points and to build the program call graph and the abstraction of its run-time heap. Then, the analyzer uses the parsed bytecode to search for potential vulnerabilities. For this purpose, we have added auto-infotainment and OBD-II checkers to the Julia framework.

Google defined a list of quality requirements[25] that Android Auto apps should respect in order to be published on the Google Play store. These requirements range from avoiding driver distractions (*e.g.*, do not display animated elements) to generic quality properties (*e.g.*, the presence of back and home buttons). Some of these requirements can be checked by static analysis. Therefore, we targeted these standard requirements as they express the issues related to infotainment systems and provide a standard benchmark to the developer community. While Julia contains already several other analyses that might be interesting for infotainment apps, we focus our efforts and extensions only on these requirements since the main goal of this work is to study and develop ad-hoc analyses for infotainment apps.

Again, as discussed earlier, OBD-II apps directly interact with safety critical car components. Thus, from a security perspective, it is important to analyze the taintedness of data that flow between the app and the car's hardware, in order to detect security vulnerabilities such as CAN injections (that is, the possibility, for an external user of an app, to inject arbitrary messages to the CAN). For this purpose, the OBD-II checker is based on Julia's taintedness analysis[26], that relies on a given set of sources and sinks. That analysis required us to specify sources and sinks for the API that is responsible for communication between the app and the car's ECU. However, almost all apps in Google Play store use custom APIs. This makes it difficult to provide a single generic solution for all OBD-II apps. Hence, we opted for the introduction of a generic framework that paves the way to the analysis of all apps. Then, we have instantiated that framework to apps relying on the OpenXC library. This library is developed by Ford Bug Labs and has been proposed as an open standard. Therefore, at present the OBD-II checker is focused on apps using

[1]This article is an extended and revised version of the manuscripts[17,18] presented at the *15th ACM International Conference on Computing Frontiers, 2018* and at the *3rd International Conference on Smart Computing and Communication, 2018.*

[2]http://checkstyle.sourceforge.net/

[3]https://scan.coverity.com/

[4]http://findbugs.sourceforge.net/

[5]https://pmd.github.io/

OpenXC, by using a generic approach that can be extended to similar libraries by instantiating their set of sources and sinks, while keeping the same core analysis unchanged.

In order to evaluate the effectiveness of our analyzer, we have collected, from Google Play store, a set of widely used infotainment apps. The results show that about 60% of the apps are potentially vulnerable, out of which 25% pose security threats related to the execution of JavaScript. Furthermore, almost all apps use a large number of unprotected intents, some easily exploitable by an intruder. In addition, we found out that 12% of the apps use weak encryption techniques, where the encryption key can be easily predicted through existing tools and techniques. Finally, the analysis results on OpenXC-based apps show possibilities of severe CAN injections and privacy violations that leak sensitive data to the web. All Android apps tested in our experiments are available from *https://github.com/amitmandalnitdgp/Android-Auto-Apps-RE-* and *http://openxcplatform.com/projects/*, so that interested readers can re-analyze them by using the Julia analyzer at *https://juliasoft.com/julia-analyzer/*. Although static analysis potentially produces false alarms (that is, warnings that indeed are not real vulnerabilities), our analyzer did not produce any in the analyzed programs. This is due to two reasons: Firstly, the infotainment checks are rather simple (*e.g.*, checking that a method is always called with a given constant value), and therefore the analysis is quite precise when dealing with these properties. Secondly, in the OBD-II analyses the Julia taint analysis did not expose any false alarm since the amount of tainted data was quite limited as we focused the analysis on the sensitive data and the user input.

The rest of the paper is organized as follows. Section 2 summarizes the attack surfaces of infotainment and OBD apps. Section 3 discusses related work. Section 4 exemplifies concrete vulnerabilities. Section 5 describes the Julia analyzer and the architecture of the new checkers that we have implemented. Section 6 presents the experiments. Section 7 concludes.

## 2 | VULNERABILITY RISKS IN INFOTAINMENT SYSTEMS

In-vehicle infotainment apps, as any software, cannot be ensured to be bug-free. While the car architecture (and in particular the CAN bus, that transmits all safety critical messages) was initially developed for operating in a closed environment with limited security concerns, it is nowadays exposed to external (sometimes, remote) attackers. Significant attention has been paid to online entertainment and to the access, from smartphones, of basic car functionalities. Namely, car manufacturers allow integration of general entertainment and utility devices into vehicles, thus raising major security concerns. Figure 1 depicts the abstract network interaction diagram between the infotainment, the OBD unit and the car's internal network. In general, the infotainment dashboard is not directly connected to the car's internal network and relies on a different gateway for that[27]. However, it can be connected to the internet via smartphone. Moreover, the OBD-II port can send data requests to crucial car components. Several attacks[28,14,13,29] have made apparent that intruders can compromise the car security by using infotainment and OBD apps. This section reports the potential vulnerability surfaces of infotainment and OBD-II-based Android apps.

### 2.1 | Infotainment Apps

In order to keep pace with the development of consumer electronics, car manufacturers introduced infotainment apps, mostly using the Android Auto open standard. Although Google suggests a comprehensive list of security and quality measures for apps, these are not always respected and attackers can exploit such holes. Android Auto apps may have common security vulnerabilities as any other regular app, such as data breaches and injections. However, Android auto introduces unique threats, that can be categorized by the type of attack, as summarized in Table 1 . '

### 2.2 | OBD Apps

The security of in-vehicle networks has additional safety implications. Modern vehicles communicate with several sensors and actuators, through an electronic bus. External devices are plugged in the bus by using an OBD-II port and communicate with the car via AT commands. Several researchers have shown that connecting OBD-II devices directly to the vehicle may compromise security of safety-critical functions. This is because the CAN bus, by design, offers no protection from manipulation[5,6]. The OBD-II port has been widely used to download diagnostic data and run tests. However, with the changing business scenario, there is market pressure to allow drivers to access the car's health data through their smartphone, also over the internet, which allows direct interaction with the car's CAN traffic. Table 2 summarizes the risks of using OBD-II apps.

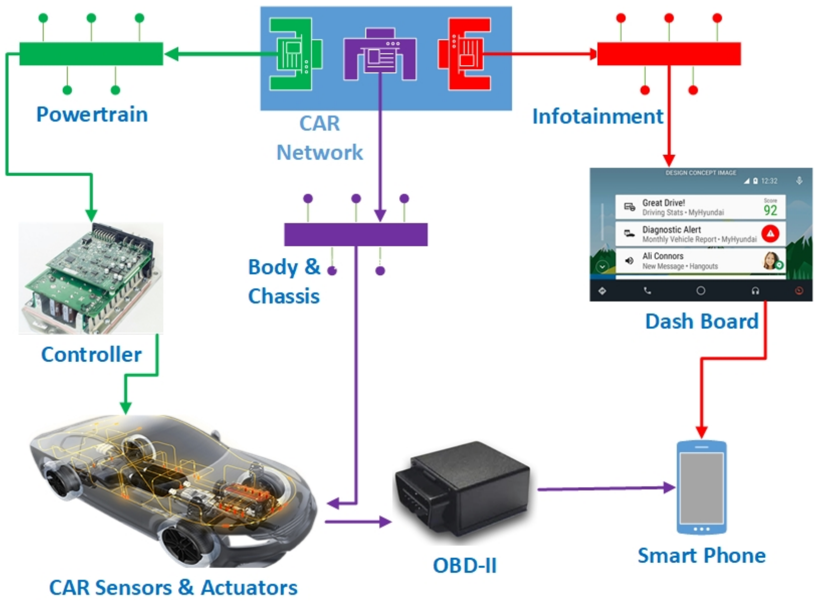


**FIGURE 1** Car components.

**TABLE 1** Threats in infotainment systems.

| **Attack Technique** | Description |
|---|---|
| **Virus/Malware** | Performs unauthorized use of infotainment functions via impersonation or attacking bugs. |
| **Authentication** | Vehicle's inherent information such as ID, equipment or authentication information code get stolen or masked for illegal use. |
| **Illegal Setting** | Vehicle data get compromised via unauthorized impersonation or attacking bugs. |
| **False Information** | Malicious apps send false messages to the infotainment system to mislead the driver or perform illegal actions. |
| **Jamming** | Malicious apps gain control of the communication route, hijack regular communication and mingle with illegal communication. |
| **Tracking** | Attackers illegally obtain vehicle information and track vehicle state information such as velocity, position and destination. |
| **Driver Distraction** | Malicious apps distract the driver by displaying images or playing audio or video. |

**TABLE 2** Threats in OBD-II-based apps.

| **Attack Technique** | Description |
|---|---|
| **CAN Injection** | Attackers send crafted CAN messages to perform dangerous operations. |
| **Compromising the OBD-II Device** | Attackers use an insecure OBD device to take control of crucial car components. |
| **Bluebugging** | Attackers take control of the OBD device when not connected to the owner's device and access crucial car components. |
| **Privacy Breach** | Intruders use OBD devices to attack mobile devices in the car and steal personal data. |
| **Tracking** | Attackers illegally obtain vehicle's information and track its state. |

## 3 | RELATED WORK

The Android-based In-Vehicle Infotainment (IVI) system has been the focus of automobile research during the last decade. Several scientific articles discuss its development[30,31], performance[32] and user experience[33,34], as well as its desirable features[35]. Only a few papers face issues related to its security and privacy.

McAfee, in partnership with Wind River and ESCRYPT, released a report called "Caution: Malware Ahead"[28], that analyzes emerging risks in automotive system security and the security of ECUs that are omnipresent in modern automotive. That study shows that an ECU connected to the infotainment system somehow facilitates cybercriminal activities: remote unlock, start and stop of a car via mobile phone; tracking of location, activities and routines of a driver; stealing of personal data via Bluetooth; disruption of the navigation system; or disabling of the emergency assistance.

Jia *et al.*[12] introduce the concept of an app-based autonomous vehicle platform that provides a framework for third-party developers. They propose an enhanced app-based vehicle design schema called AVGUARD to address safety and security issues. This primarily focuses on mitigating the threats posed by the use of untrusted code, by leveraging the theories of vehicle evaluation field and program analysis techniques. Moreover, the study sketches a guideline and suggests good practices for the improvement of future automotive apps. Paupiah *et al.*[36] and Bordonali *et al.*[37] discuss the various security threats posed by the use of Android-based infotainment systems. Kim *et al.*[38] analyze an access control for IVI and propose the *restricted execution environment system*, to protect a mobile handset connected to the car. That is a malware detection system that analyzes programs and provides mobility. Beside this, Nisch[39] provides an insight into different security vulnerabilities of the automotive ECU. It carries out a detailed analysis of the threats to various ECU units, such as the tire pressure monitoring system, the GPS, the keyless entry system, the OBD, the audio system, the Bluetooth connectivity and the mobile phone interface. Results show that these units individually or collectively induce serious security threats to the car.

Researchers in both industry and academia have shown possible attacks to safety-critical automobile components. Mazloom *et al.*[29] analyzes vulnerabilities of apps, protocols and underlining IVI, due to a smartphone connected to the car infotainment system. For this purpose, they consider an IVI system that supports the MirrorLink protocol in a 2015 model of a major automotive manufacturer. They show that vulnerabilities in that protocol can help an attacker send malicious messages to the vehicle's CAN network, from a connected smartphone.

Modern vehicles connect their embedded hardware, such as sensors and actuators, through an electronic bus. External devices can be plugged to the bus by using an OBD-II port and by sending AT commands. The most popular device is ELM327, whose AT commands are publicly available online[6]. The CAN bus protocol is the most widely adopted standard bus both in USA and Europe. It was designed to be fast and robust. Hence, for instance, communication is unauthenticated and unencrypted. However, the CAN is nowadays connected to the driver and the external world through smartphones and tablets plugged in through Bluetooth or USB ports. This paves the way to security attacks to the car and to privacy leaks of the transferred data, as Checkoway *et al.*[8], Koscher *et al.*[40] and Avatefipour *et al.*[41] show. Such articles exemplify how an attacker can get complete control over the vehicle's systems[8]. More recently, authentication has been added to protocols[42], but this increases the latency time, does not completely solve the injection issues and does not apply to legacy systems.

In 2015, Miller *et al.*[14] demonstrated a serious security vulnerability in cars, which obliged the manufacturer to recall 1.4 million vehicles. It was possible to send carefully crafted messages to the CAN in order to remotely hack into a car, immobilize it in highway traffic, cause unintended acceleration and even disable its brakes. To that goal, they used the vehicle infotainment system to access the ECU that sends legitimate commands to other ECU components. To cope with such attacks, Schweppe *et al.*[13] present an architecture capable of monitoring data flow through the car's CAN. This approach enhances vehicle security by using taint tracking tools along with a security framework able to dynamically tag data flows within or among control units. They also implement a prototype to prevent damage to the on-board system, through buffer overflows. Furthermore, they show the applicability of transport tags among network nodes by extending the communication payload. However, their overhead is very high because of taint tracking. This makes their technique unsuitable for real-time environments.

Beside security, little attention has been dedicated to the privacy of the drivers. De Graaff *et al.*[11] discuss the enforcement of a higher level of privacy by using cryptographic techniques. They identify technical requirements that lead to the construction of a homomorphic cryptography solution with semi-trusted third-party architecture, thus eliminating many disadvantages in communication channels. Jaisingh *et al.*[43] provide an overview of how personal information flows through typical infotainment and telematics systems. They also identify potential privacy threats to drivers and provide security recommendations.

[6]https://www.sparkfun.com/datasheets/Widgets/ELM327_AT_Commands.pdf

## 4 | EXPLOITABLE VULNERABILITIES

The Android ecosystem is a complex open network of collaborating companies. It uses more than 170 open-source projects, including Google's operating system. Moreover, hardware manufacturers and network providers customize Android to their requirements. This leaves the system in a potentially vulnerable state. In particular, the Application Programming Interface (API) of Android Auto allows developers to interact with the infotainment system. Often the API designer had a particular protocol or API call sequence in mind to ensure the security and reliability of the system, but an attacker could break that intended model. In particular, our analyses target the following vulnerabilities of Android Auto infotainment and OBD-II-based apps.

### 4.1 | Android Auto Apps

Google defines a set of quality requirements for Android Auto apps, whose violation induces the following vulnerabilities. What follows is a comprehensive list of vulnerabilities that have guided the development of our checks. The experimental results in Section 6 have actually found only some of them in real-world apps.

1. *External File Access Detection*: Files created in the external storage, such as SD cards, are globally readable and writable. Therefore, app data should not hold sensitive information using external storage, that can be removed by the user and modified by any malicious app. Furthermore, apps using external storage should perform input validation when handling data from external storage, as it could contain executables or data from untrusted sources, that cause damage to the car. The code snippet in Listing 1 shows an untrusted use of an external directory by calling method `Context.getExternalCacheDir()`.

   Listing 1: Untrusted use of an external directory.

   ```
   public static File getDiskCacheDir(Context c) {
      File dir = c.getExternalCacheDir();
      if (dir == null)
         dir = c.getCacheDir();
      return dir;
   }
   ```

2. *Usage of WORLD_WRITEABLE* : By default, Android enforces that only the app that created a file on the internal storage can access it. However, some apps do use MODE_WORLD_WRITEABLE or MODE_WORLD_READABLE for files, thus bypassing that restriction. They also exploit the ability to load and control the data format. With world writable or readable enabled, malicious apps can tamper and/or steal private information from the car's dashboard or from the smartphone. The code snippet in Listing 2 shows this practice where MODE_WORLD_WRITEABLE is passed to `openFileOutput`.

   Listing 2: Vulnerable use of WORLD_WRITEABLE.

   ```
   File f = new File(getFilesDir(), "filename.ext");
   f.delete();
   FileOutputStream fos = openFileOutput("filename.ext", Context.MODE_WORLD_WRITEABLE);
   fos.close();
   File f = new File(getFilesDir(), "filename.ext");
   ```

3. *Encryption Function*: In order to protect sensitive data, local files are often encrypted with a key not directly accessible to the app. Keys can be placed in a keystore and protected with a password. The code snippet in Listing 3 shows the usage of a keystore: if `PasswordProtection()`, `load()` and `getPrivateKey()` get tainted, then the security of the system is compromised, as well as the key stored in the system for later use. These keys are not erased from memory after their use.

   Listing 3: Malicious usage of a keystore.

   ```
   KeyStore ks = KeyStore.getInstance("JKS");
   char[] password = getPassword();
   try (FileInputStream fis = new FileInputStream("keyStoreName")) {
     ks.load(fis, password);
   }
   // get private key
   KeyStore.ProtectionParameter protParam = new KeyStore.PasswordProtection(password);
   KeyStore.PrivateKeyEntry pkEntry = (KeyStore.PrivateKeyEntry) ks.getEntry("privateKeyAlias", protParam);
   ```

```
PrivateKey myPrivateKey = pkEntry.getPrivateKey();
// save secret key
javax.crypto.SecretKey mySecretKey = ...;
KeyStore.SecretKeyEntry skEntry = new KeyStore.SecretKeyEntry(mySecretKey);
ks.setEntry("secretKeyAlias", skEntry, protParam);
// store in the keystore
try (FileOutputStream fos = new FileOutputStream("newKeyStoreName")) {
   ks.store(fos, password);
}
```

Moreover, Android implements the *SHA1PRNG* algorithm in *SecureRandom* instances. That algorithm is cryptographically weak, since it has been shown that its *random* sequences, in binary form, tends to 0's, specifically for some seeds. A common but potentially harmful use of this algorithm is the creation of encryption keys by using a password as a seed[44]. Due to some implementation issues, the key becomes deterministic if the seed is generated with *SecureRandom*.

4. *Unprotected Intents*: Content providers are a structured storage mechanism that can be limited to a given app (if the Android manifest contains `android:exported=false`) or exported to all apps (if it contains `android:exported=true`). If intents are used to export data towards other apps, a permission for reading and/or writing must be used for protection. However, developers do not always use such protection mechanism. This may lead to intent leakage and to other serious security issues. Note that permissions can also be added and removed programmatically, as shown in the code snippet in Listing 4, that calls method `PackageManager.addPermission`:

Listing 4: Dynamic addition of a dangerous permission.

```
PermissionInfo pi = new PermissionInfo();
pi.name = myCustomPermission;
pi.labelRes = R.string.permission_label;
pi.protectionLevel = PermissionInfo.PROTECTION_DANGEROUS;
PackageManager packageManager = getApplicationContext().getPackageManager();
packageManager.addPermission(pi);
```

5. *Use of WebView*: Android's WebView class implements a browser widget that supports HTML and JavaScript. Its improper use can lead to major web security issues such as cross-site-scripting (XSS) or JavaScript injection. In this regard, method `WebView.setJavaScriptEnabled()` paves the way to XSS. Hence, if an app uses the code snippet in Listing 5, then one should check the taintedness of the rendered page.

Listing 5: Improper use of a WebView instance.

```
WebView myWebView = (WebView) findViewById(R.id.webView);
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

Moreover, method `WebView.addJavaScriptInterface()` lets JavaScript invoke methods of the app itself. In that case, according to Google's recommendation, only trusted web pages should be visited or otherwise untrusted JavaScript can invoke Android methods within the app. In particular, only JavaScript contained within the apk should be run. The code snippet in Listing 6 shows an unsafe use of this method.

Listing 6: Untrusted use of JavaScript.

```
public class JavaScriptAttack extends Activity {
   protected void onCreate(Bundle savedInstanceSTate){
      super.onCreate(savedInstanceSTate);
      setContentView(R.layout.activity_jscript_attack);
      WebView wv = new WebView(getApplicationContext());
      wv.getSettings().setJavaScriptEnabled(true);
      wv.addJavaScriptInterface(new jsInvokeclass(), "attack");
      wv.loadUrl("http://www.malware.com/atk.html");
   }
}
```

6. *GPS Location Detector*: Method `WebChromeClient.onGeolocationPermissionsShowPrompt()` is used by class WebView to determine if it can disclose the user's location to JavaScript. Its implementation should seek permission

from the user. The code snippet in Listing 7 instead grants that permission always, hence compromising the car's location service.

Listing 7: Leakage of GPS location.

```
webView.setWebChromeClient(new WebChromeClient() {
   public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {
      callback.invoke(origin, true, false);
   }
}
```

7. *Background Download*: In order to download a file, if the storage location is not explicitly set, the programmer uses the `DownloadManager.openDownloadedFile()` method, with the ID value stored in preferences, to get a `ParcelFileDescriptor` that can be turned into an input stream. Furthermore, without a specific destination, downloaded files stay in the shared download cache. In this case, the system retains the right to delete them at any time to reclaim space. This leaves the app in a vulnerable state, since shared data can be freely accessed. Thus, it is necessary to check the taintedness of the functions in the code snippet of Listing 8, to avoid potential security breaches.

Listing 8: Untrusted background download.

```
Request.setDestinationInExternalFilesDir(): Set the destination to a hidden directory on external storage
Request.setDestinationInExternalPublicDir(): Set the destination to a public directory on external storage
Request.setDestinationUri(): Set the destination to a file Uri located on external storage
```

8. *Media Autoplay*: According to the security standards set by Google, "media apps must not autoplay on the startup of Android Auto or without user initiated action to select the app or app media"[25], since this can distract the driver. Thus, one should avoid code as in Listing 9, that autoplays media files through the call `WebSettings.setMediaPlaybackRequiresUserGesture(false)`.

Listing 9: Malicious use of media autoplay.

```
public class myProject extends CordovaActivity {
   public void onCreate(Bundle savedInstanceState){
      super.onCreate(savedInstanceState);
      super.init();
      super.loadUrl(Config.getStartUrl());
      WebSettings ws = super.appView.getSettings();
      ws.setMediaPlaybackRequiresUserGesture(false);
   }
}
```

9. *CarMode*: Google Play store lists Android Auto apps in a separate category. They should explicitly call `UiModeManager.enableCarMode("true")` to qualify as Android Auto apps. This prevents users from installing apps not suitable for cars.

10. *Voice Commands*: Android Auto apps should allow users to control audio content playback with voice actions. This provides a hands-free experience to the driver, who can concurrently drive and interact with the infotainment. To get voice-enabled playback controls, Android Auto apps must turn on the hardware controls by setting, in the app's `MediaSession` instance, the flags reported in the code snippet in Listing 10.

Listing 10: Usage of voice commands.

```
mSession.setFlags(MediaSession.FLAG_HANDLES_MEDIA_BUTTONS|MediaSession.FLAG_HANDLES_TRANSPORT_CONTROLS);
```

11. *Displaying Online Images*: According to the standards set by Google, Android Auto apps should not display any image advertisement, that could distract the driver. Online images are usually accessed through some web API or online service. Thus, apps should not use such services. The code snippet in Listing 11 shows an example.

Listing 11: Displaying online images.

```
public ImageViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
View view = LayoutInflater.from(OnlineImageActivity.this).inflate(R.layout.image_item, parent, false);
return new ImageViewHolder(view);
}
```

12. *Displaying HTML*: The simplest case is to display HTML or images by supplying their URL to a WebView by calling `WebView.loadUrl`. This can lead to injections, especially when JavaScript is enabled. The code snippet in Listing 12 shows an example.

Listing 12: Displaying HTML.

```
public class MyActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      WebView webview = new WebView(this);
      webview.getSettings().setJavaScriptEnabled(true);
      webview.loadUrl("http://www.malware.com/");
      setContentView(webview);
   }
}
```

Furthermore, a malicious program can exploit the `WebViewClient.shouldOverrideUrlLoading()` callback to intercept, monitor and log user activities. This is a serious privacy breach. The code snippet in Listing 13 shows an example.

Listing 13: Vulnerable usage of WebViewClient.

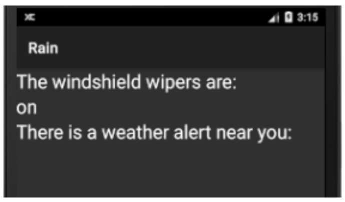
```
public class MyActivity extends Activity {
   String mCurrentUrl="";
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      WebView webview = new WebView(this);
      webview.getSettings().setJavaScriptEnabled(true);
      webview.setWebViewClient(mClient);
      webview.loadUrl("http://www.malware.com");
      setContentView(webview);
   }

   private WebViewClient mClient = new WebViewClient() {
     public boolean shouldOverrideUrlLoading(WebView wv, String url) {
       // the url visited by the user is saved in a field, where it remains visible
       mCurrentUrl = url;
       Log.i("mCurrentUrl", String.valueOf(mCurrentUrl));
       view.loadUrl(url);
       return true;
     }
   }
}
```

13. *Media Advertisement*: Google dictates that Android Auto apps should not display notifications, hence their media metadata key `android.media.metadata.ADVERTISEMENT` should be set to 1. The code snippet in Listing 14 keeps the provision of displaying the notification open, instead.

Listing 14: Improper media advertisement.

```
public static final String EXTRA_METADATA_ADVERTISEMENT = "android.media.metadata.ADVERTISEMENT";
public void onPlayFromMediaId(String mediaId, Bundle extras) {
   MediaMetadata.Builder builder = new MediaMetadata.Builder();
   if (isAd(mediaId))
      builder.putLong(EXTRA_METADATA_ADVERTISEMENT, 0);
   mediaSession.setMetadata(builder.build());
}
```


Rain

The windshield wipers are:

on

There is a weather alert near you:

**FIGURE 2** The Rain Monitor app.

## 4.2 | OBD-II Apps

This section describes the most common security vulnerabilities of OBD-II apps. To exemplify these issues, a few open-source apps using the OpenXC library have been considered. As discussed earlier, we restrict our findings to the OpenXC API only, but the approach can be easily extended to other OBD-II APIs.

1. *Privacy Breach*: OBD-II apps can collect sensitive information from different sensors and actuators. Malicious apps may send such information to a remote server, by using web services, where the collected data is insecurely transmitted or stored. Furthermore, they could track and analyze such data without proper user consent.

   For instance, the Rain Monitor app[7] (Figure 2 ) uses OpenXC to collect data regarding location, windshield status and speed in method `run()`, and sends this information to a remote web service, without any encryption or authentication in method `uploadWiperStatus`. The code snippet in Listing 15 reports where this occurs. The status of the HTTP request and of the windshield gets also logged at the end of method `uploadWiperStatus`. This flow is caught by Julia's flow analysis, hence detecting privacy violations.

Listing 15: A snippet of code from the Rain Monitor app. Sensitive car data is sent to the internet and logged.

```
public class CheckWipersTask ... {
   private final String WUNDERGROUND_URL = "http://www.wunderground.com/weatherstation/VehicleWeatherUpdate.php";
   private VehicleManager mVehicle;
   ...
   public void run() {
      // get messages from the CAN by means of the OpenXC API library
      Latitude latitude = (Latitude) mVehicle.get(Latitude.class);
      Longitude longitude = (Longitude) mVehicle.get(Longitude.class);
      WindshieldWiperStatus wiperStatus = (WindshieldWiperStatus)
      mVehicle.get(WindshieldWiperStatus.class);
      ...
      boolean wiperStatusValue = wiperStatus.getValue().booleanValue();
      ...
      uploadWiperStatus(latitude, longitude, wiperStatus);
   }

   private void uploadWiperStatus(Latitude latitude, Longitude longitude, WindshieldWiperStatus wiperStatus) {
      StringBuilder uriBuilder = new StringBuilder(WUNDERGROUND_URL);
      int wiperSpeed = 0;
      boolean wiperStatusValue = wiperStatus.getValue().booleanValue();
      if (wiperStatusValue)
        wiperSpeed = 1;
      // construct the url with information to send
      uriBuilder.append("?wiperspd=" + wiperSpeed);
      uriBuilder.append("&lat=" + latitudeValue);
      uriBuilder.append("&lon=" + longitudeValue);
      String finalUri = uriBuilder.toString();
      ...
      HttpClient client = new DefaultHttpClient();
      // send the CAN data on the internet and receive an ack back
      HttpGet request = new HttpGet(finalUri);
      HttpResponse response = client.execute(request); // line 111
      int statusCode = response.getStatusLine().getStatusCode();
      if (statusCode != HttpStatus.SC_OK)
         Log.w(TAG, "Error " + statusCode + " while uploading wiper status"); // line 114
      else
         Log.d(TAG, "Wiper status (" + wiperStatus + ") uploaded successfully"); // line 117
```

[7]https://github.com/openxc/rain

```
   }
}
```

The code snippet in Listing 16 reports another portion of code from the same app. It reads the car position from the CAN and logs it. Hence, anybody having access to the logs can reconstruct the movements of the vehicle, a clear privacy issue. At the end, this code builds a URL by using latitude and longitude. This is an example of sensitive data flowing into an internet address, although possibly inherent to the task performed by this app.

Listing 16: A snippet of code from the Rain Monitor app. Sensitive car data is logged and used to build a URL address.

```java
public class FetchAlertsTask extends TimerTask {
   private final String TAG = "FetchAlertsTask";
   private final String API_URL = "http://api.wunderground.com/api/dcffc57e05a81ad8/alerts/q/";
   ...
   public void run() {
      Latitude latitude = (Latitude) mVehicle.get(Latitude.class);
      Longitude longitude = (Longitude) mVehicle.get(Longitude.class);
      double latitudeValue = latitude.getValue().doubleValue();
      double longitudeValue = longitude.getValue().doubleValue();
      ...
      Log.d(TAG, "Querying for weather alerts near " + latitudeValue + ", " + longitudeValue); // line 68
      ...
      StringBuilder urlBuilder = new StringBuilder(API_URL);
      urlBuilder.append(latitudeValue + ",");
      urlBuilder.append(longitudeValue + ".json");
      URL wunderground = new URL(urlBuilder.toString()); // line 76
      ...
   }
   ...
}
```

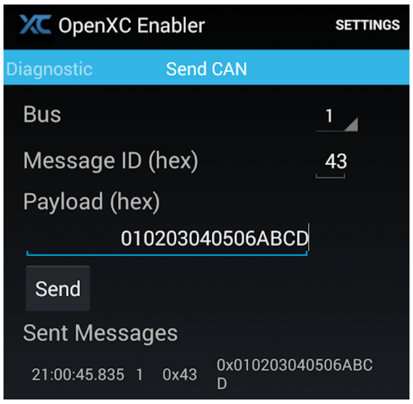
2. *Injecting Data into the CAN*: Apps may provide direct access to the CAN bus of a vehicle with minimum or no authentication. Furthermore, one can use such insecure channel to inject crafted CAN messages to obtain various levels of physical control over the vehicle.

   OpenXC Enabler [8] is a tutorial app meant to test and document most functionalities of OpenXC. It is bundled with the library itself. It allows one to send arbitrary messages to the CAN, as shown in Figure 3 . The user can insert the CAN bus number to use, the ID of a target sensor or actuator and a value containing multiple CAN signals for it, in JSON format, such as: *{"bus": 1, "id": 43, "value": "0x0102003040506ABCD"}*. That message gets delivered to the sensor or actuator, which will react accordingly. This app features a flow of information from user input into the CAN, that is, an injection of data into the CAN. Such injection might be exploited to send arbitrary messages to the CAN, and different ECU might react to such messages (for instance, by unlocking the car). The code snippet in Listing 17 reports where data is collected from GUI widgets, packed into a message and sent to the CAN. Again, no encryption nor authentication is used.

Listing 17: A snippet of code from the OpenXC Enabler app. User data is read from GUI widgets and sent to the CAN.

```java
public class SendCanMessageFragment ... {
   private void onSendCanMessage(Spinner busSpinner, EditText idView, EditText payloadView) {
      ...
      // construct a message by attaching every parameter coming from the user
      CanMessage message = new CanMessage(
          Integer.valueOf(busSpinner.getSelectedItem().toString()),
          Integer.valueOf(idView.getText().toString(), 16),
      ByteAdapter.hexStringToByteArray(payloadView.getText().toString()) );
      // send the message to the CAN
      mVehicleManager.send(message); // line 110
      ...
   }
}
```

[8]https://play.google.com/store/apps/details?id=com.openxcplatform.enabler&hl=en



Bus 1

Message ID (hex) 43

Payload (hex)

010203040506ABCD

Sent Messages

21:00:45.835 1 0x43 0x010203040506ABCD

**FIGURE 3** The OpenXC Enabler app.

The code snippet in Listing 18 of the same app reports where data coming from Android preferences, hence chosen by the user, is concatenated into `combinedAddress` and then logged.

Listing 18: A snippet of code from the OpenXC Enabler app. Data coming from Android preferences is concatenated into combinedAddress and then logged.

```
public abstract class VehiclePreferenceManager
   ...
   protected String getPreferenceString(int id) {
     return getPreferences().getString(mContext.getString(id), null);
   }
   ...
}

   public class NetworkPreferenceManager extends VehiclePreferenceManager {
      private final static String TAG = "NetworkPreferenceManager";
      ...
      private void setNetworkStatus(boolean enabled) {
         Log.i(TAG, "Setting network data source to " + enabled);
         if (enabled) {
            String address = getPreferenceString(R.string.network_host_key);
            String port = getPreferenceString(R.string.network_port_key);
            String combinedAddress = address + ":" + port;
            if (address == null || port == null || !NetworkVehicleInterface.validateResource(combinedAddress)) {
            String error = "Network host URI (" + combinedAddress + ") not valid -- not starting network data
                 source";
            Log.w(TAG, error); // line 53
            ...
         }
      }
   }
}
```

3. *CAN Data Flows into the Internal Logic of Apps*: Several apps perform crucial tasks based on data received from the car's ECU via OBD-II devices. CAN data that flows through the internal logic requires proper sanitization and should not be altered, since that might mislead the driver. Thus, ensuring security and integrity of such apps is extremely important.

   The Shift Knob OpenXC app [9] is "a replacement for a manual transmission shift knob that adds haptic and visual feedback to help drivers shift appropriately". It monitors vehicle information, coming from the CAN, and provides suggestions to the driver about good driving style, by vibrating the shift knob. For instance, it suggests when it is the right moment to shift. Clearly, this app accesses CAN data, but this is then used in a controlled way, only inside the logic of the app. Data is also reported in the UI (Figure 4 ), but never divulged through external means, such as the internet. As a consequence, this app does not feature any injection. The code snippet in Listing 19 shows a portion of code from this app. It defines a listener that feeds, into a UI widget, CAN data about the speed of the car. That data does not propagate further.

[9]http://openxcplatform.com/projects/shift-knob.html

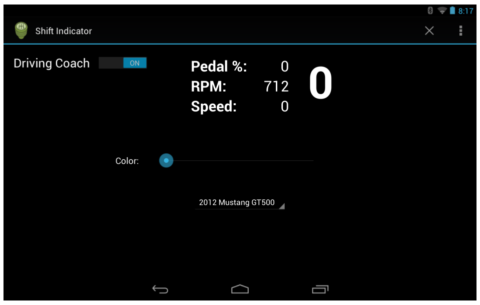


**FIGURE 4** The Shift Knob app.

Listing 19: A snippet of code from the Shift Knob app. Sensitive car data coming from the CAN flows into the listener and then into a widget of the same app, but it is not sent outside the device.

```
Measurement.Listener mSpeedListener = new Measurement.Listener() {
   public void receive(Measurement measurement) {
      final VehicleSpeed updated_value = (VehicleSpeed) measurement;
      mVehicleSpeed = updated_value.getValue().doubleValue();
      runOnUiThread(new Runnable() {
         public void run() {
            // send vehicle speed with 1 decimal point
            mVehicleSpeedView.setText("" + Math.round(mVehicleSpeed * 10) / 10);
         }
      });
   }
};
```

The Night Vision app [10] "adds night vision to a car with off-the-shelf parts. The webcam faces forward from the dashboard and uses edge detection to detect objects on the road in the path of the vehicle". The app monitors the car's headlamps. As soon as they are turned on, the app is launched and opens a WebView that identifies objects intercepted by the webcam (Figure 5 ). This app uses OpenXC only for listening to the headlamps status. This is done by the listener shown in the code snippet in Listing 20. When the headlamps are turned on, this listener starts the main activity of the app. Sensitive data (the state of the headlamps) is only used inside the logic of the app and does not flow outside the device. Hence, this app does not feature any injection.

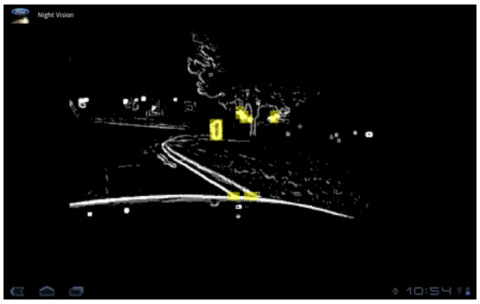


**FIGURE 5** The Night Vision app.

Listing 20: A snippet of code from the Night Vision app. Sensitive car data coming from the CAN flows into the listener, but it is not sent outside the device.

```
Measurement.Listener mHeadlampListener = new Measurement.Listener() {
```

[10]http://openxcplatform.com/projects/nightvision.html

```
   public void receive(Measurement measurement) {
      final HeadlampStatus status = (HeadlampStatus) measurement;
      mHandler.post(new Runnable() {
         public void run() {
         if (status.getValue().booleanValue()) // are headlamps on?
            if (!NightVisionActivity.isRunning()) {
               Intent intent = new Intent(VehicleMonitoringService.this, NightVisionActivity.class);
               intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
               VehicleMonitoringService.this.startActivity(intent);
            }
         else
            sendBroadcast(new Intent(ACTION_VEHICLE_HEADLAMPS_OFF));
         }
      });
   }
};
```

4. *CAN Data Stored into a Database*: Apps may store different sensor data of ECUs in databases for various analysis purposes. This can lead to SQL-injection attacks and compromise the reliability of the apps, if not properly sanitized. Thus, one must ensure that data flowing from CAN to database is not tainted.

   Consider for instance the MPG app [11]. It "takes information such as trip distance, trip length, gas consumption and vehicle speed to determine your current fuel usage over a drive. After a drive is completed, your fuel consumption/fuel efficiency is calculated and saved to a local SQLite database". The same information is shown to the screen (Figure 6 ). The same description of the app indicates that it builds a flow of information from sensitive data coming from the CAN to a database. As such, this could be a dangerous data flow, leading to an SQL injection. The code snippet in Listing 21 shows where the data is stored in the database. The arguments of method `saveResults()` are tainted, since they are computed from CAN data. However, they have type double, hence it is hard to use them to build an SQL-injection attack string. Moreover, the `insertOrThrow()` method of the Android library guarantees that the elements inside the ContentValues object undergo sanitization before being used for the SQL query. That is, they are consistently escaped so that no SQL-injection attack can be built from them. This means that no SQL-injection can occur here.

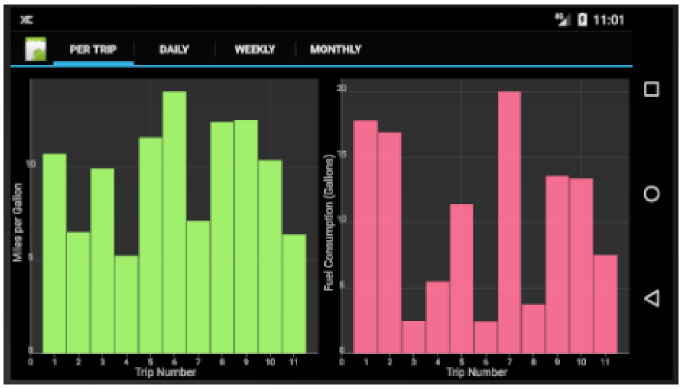


**FIGURE 6** The MPG app.

Listing 21: A snippet of code from the MPG app. Sensitive car data coming from the CAN flows into the database.

```
public class DbHelper extends SQLiteOpenHelper {
   public void saveResults(double dist, double fuel, double mileage, double start, double end) {
      double length = (end-start) / (1000*60);
      ContentValues values = new ContentValues();
      values.put(C_DISTANCE, dist);
      Timestamp time = new Timestamp((long) start);
      String stime = time.toString();
```

[11] https://github.com/openxc/mpg

```
        values.put(C_TIME, stime);
        values.put(C_LENGTH, length);
        values.put(C_FUEL, fuel);
        values.put(C_MILEAGE, mileage);
        SQLiteDatabase db = getWritableDatabase();
        db.insertOrThrow(TABLE, null, values);
    }
}
```

## 5 | A STATIC ANALYZER FOR ANDROID AUTO APPS

This section describes our new static analyzer for Android Auto apps. Its infotainment and OBD-II checker relies on heap and call graph abstractions performed by Julia[45], but it implements completely novel checks targeting the possible vulnerable points. The main novelties are the extension of the Julia analyzer to Android Auto apps and a new procedure for detecting entry points for the analysis of such apps, inspired by the vulnerabilities discussed in Section 4. Moreover, a framework for analyzing the vulnerabilities of OBD-II apps is presented.

### 5.1 | The Julia Static Analyzer

The Julia static analyzer applies abstract interpretation to the analysis and verification of Java bytecode[22]. It is based on the theoretical concepts of denotational and constraint-based static analysis through abstract interpretation. The Julia library provides a representation of Java bytecode suitable for abstract interpretation. This representation uses state transformers and also generates a call graph modeling exceptional paths. Julia simplifies the Java bytecode through explicit type information available about their operands, the stack elements and locals. Further, Julia also provides the exact implementation of the fields or methods that are accessed or called. Many analyses have been implemented on top of the Julia library. These verify the absence of a large set of typical errors in software, such as null-pointer accesses, non-termination, wrong synchronization and injection vulnerabilities.

### 5.2 | Architecture of the System

Figure 7 shows a schematic diagram of the Android Auto static analyzer. Apps are first reverse-engineered with `dex2jar`[23] and `apktool`[24]. These tools extract the app manifest and jar files from the apk. The manifest is then used to determine the entry points for analyzing the Java bytecode. The Android Auto checker is implemented for Android API 25 and applied to the parsed bytecode to detect vulnerabilities. Thus, the selection of entry points and the reconstruction of the call graph and of the heap structure plays a crucial role. The infotainment checker is based on the vulnerabilities discussed in Section 4.1, while the OBD checker is based on the principles discussed in Section 4.2. The following subsections describe the crucial components of this static analyzer.

#### 5.2.1 | Entry Points

In static analysis, the entry points are a crucial element for the soundness and coverage of the analysis, as they determine the coverage of the call-graph the analysis relies on when building the semantic model of the system. The core Julia library is built for generic Java applications, where Julia starts the analysis of a program from its main method. This becomes more complex for Android-based code, as the entire program works through multiple event handlers that may be invoked by the Android framework. Therefore, the Android Auto static analyzer starts the analysis from all such handlers. Moreover, every Android app contains an *AndroidManifest.xml* file, that describes important properties, such as program structure, permissions and user interface parameters. To get complete information about the event handlers, the analyzer must also consider how they receive input at run time, by looking at XML files, such as layout files, that are dynamically inflated.

Beside the generic event handlers in activities, services, broadcast receivers, content providers, WebView services, FileStorage and DownloadManager, special attention has been given to the MediaBrowser and Messaging services as Android Auto currently supports audio playback and messaging for music. Namely, Android Auto browses audio track listings by using the MediaBrowser service. Audio apps must declare this service in their manifest. This allows the dashboard system to discover this service and connect to the app. Figure 8 shows the extraction process of the MediaBrowser service

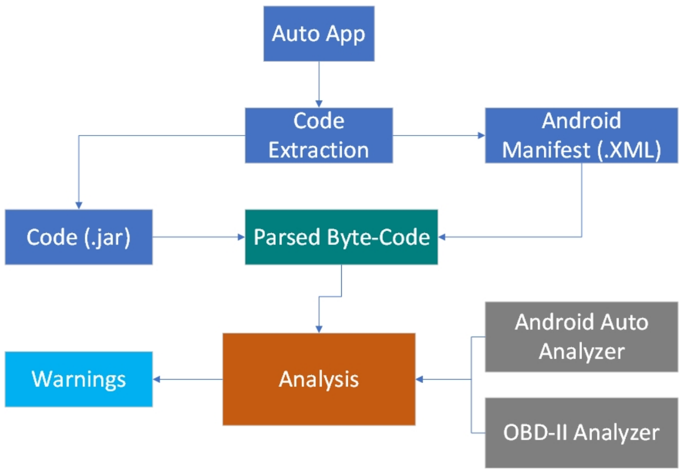


**FIGURE 7** System architecture of the Android Auto static analyzer.

class from the Android Auto apk. Here, the analyzer first looks in the manifest for the class responsible for enabling the Android.media.browse.MediaBrowserService. Then, this class is used as an entry point to parse the bytecode. Moreover, classes responsible for creating a *MediaSession* service are also considered as entry point.

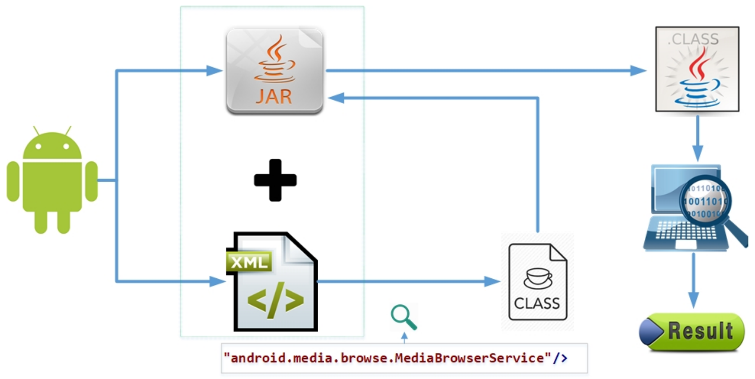


**FIGURE 8** Analysis entry points: audio playback classes.

Similarly, for the messaging services all receiver classes defined in the manifest are collected. However, the majority of these are not responsible for sending or receiving messages. According to the Android Auto specifications, classes responsible for sending and receiving messages must extend BroadcastReceiver. Thus, to filter the unnecessary receiver classes, their superclasses are checked. If the superclass is BroadcastReceiver, then it is considered as an entry point. A similar process is applied for locating other kinds of activities and services.

### 5.2.2 | Infotainment Checker

The identified entry points help building a semantic model of the app execution. The Android Auto checker searches for vulnerable API calls in the production code. If such calls are found, then it is necessary to check the taintedness of the implementation. For that, the JVM stack for the call is analyzed and all producers of the values passed to the call are traced in order to check if unfiltered or unrestrained inputs occur. If arguments are found to be unsanitized, then there is a risk of possible attacks. Figure 9 shows the process.

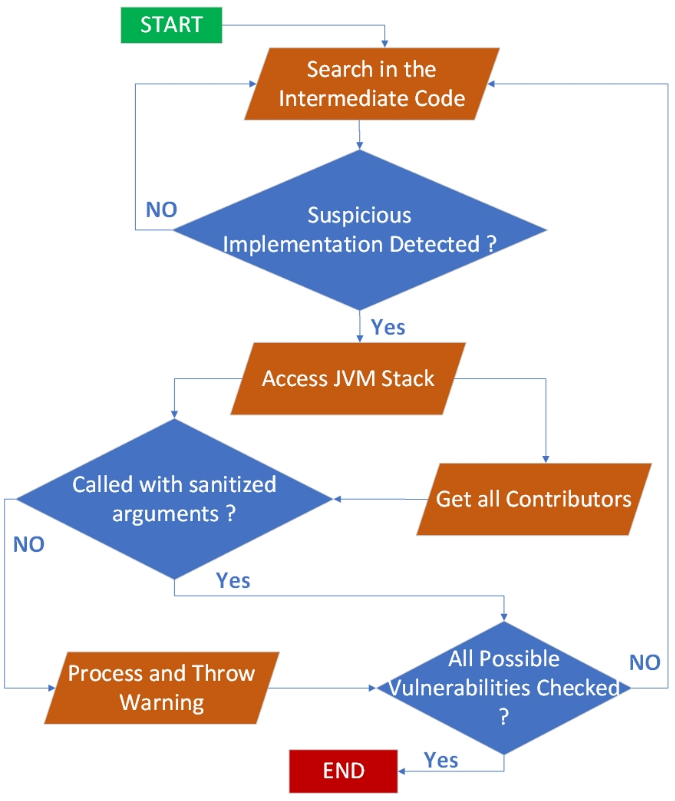

START
Search in the Intermediate Code
Suspicious Implementation Detected ?
NO
Yes
Access JVM Stack
Called with sanitized arguments ?
Get all Contributors
NO
Yes
Process and Throw Warning
All Possible Vulnerabilities Checked ?
NO
Yes
END

**FIGURE 9** Working principle of the Android Auto analyzer.

### 5.2.3 | OBD-II Checker

As discussed earlier, almost every OBD-II app uses its own custom API to communicate with the car's ECU. Thus, unlike infotainment apps, it is very difficult to provide a single solution applicable for all OBD-II apps. For this purpose, this section provides a framework for the static analysis of OBD-II apps and instantiates it to the OpenXC API. It is based on a taint analysis, instantiated to that library, as described below. Figure 10 shows the working principle of OBD-II checker.

1. *Instantiation of the OBD-II Library*: The OBD-II checker implemented in Julia uses a dictionary of sources and sinks specific for OBD-II API. Sources include methods accessing sensitive information, about the user or device, or reading data from UI widgets; sinks include methods for logging, database or network manipulation, specific to Android. By default, Julia comes with a specification of the most used sources and sinks of the standard Android runtime. The analysis of a source forces the corresponding Boolean variable to be true. At each sink, the analyzer checks if the corresponding Boolean local variable is definitely false. If that is the case, no flow of tainted data into that sink is possible; otherwise, it issues a warning, reporting a potential flow of tainted data into the sink. This approach uses a single Boolean mark for all sources. Hence, with this technique, it is inherently impossible to distinguish different origins of tainted data. However, this limitation justifies the scalability of the technique.

   The code snippet in Listing 22 reports the methods of the OpenXC library that either produce (sources) sensitive, tainted data, that should not flow into sensitive locations, or receive (sinks) data that must not be tainted, since it might flow into the CAN device. This information is in the mind of the library developers, and it is not apparent in code. In order to use the taint analysis of Julia, that information must be first made explicit, in a format that Julia can understand. Currently, Julia allows one to instantiate its taint analysis with a specification of further sources and sinks, given either as an XML file or as annotated interfaces. This article exploits the latter possibility. Namely, the annotated interfaces like code snippet of Listing 22 are provided to Julia before the analysis. Such interfaces reflect the methods where either sources (@UntrustedDevice) or sinks (@DeviceTrusted) occur, or both. For instance, methods `get()` receive a parameter that specifies the kind of information that must be read from the CAN. Hence, that parameter must not be freely in control of the user of the application, or otherwise she might be able to build an injection into the CAN device. Hence, it is a

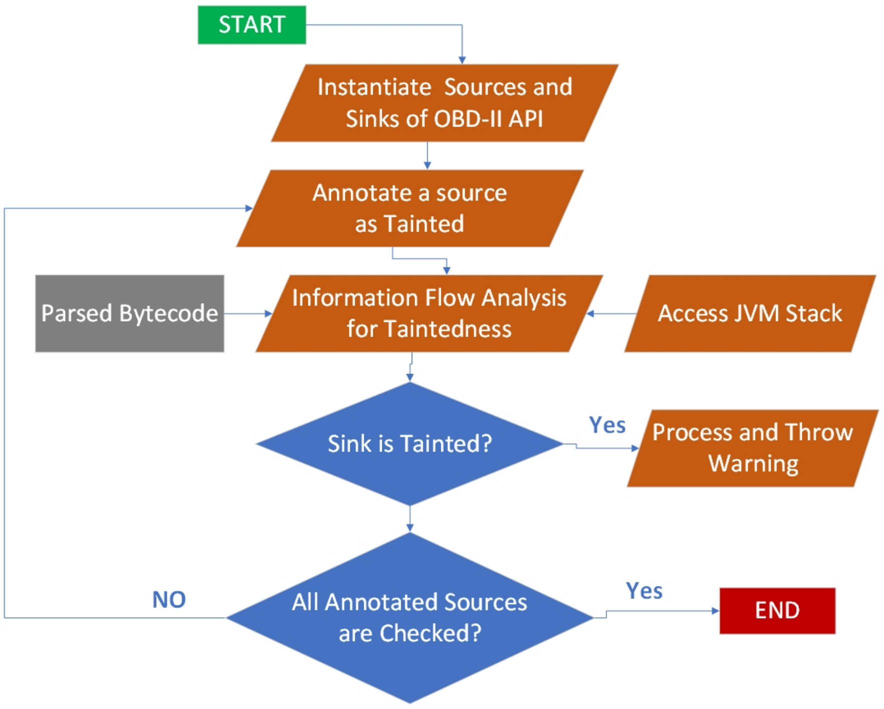

START
Instantiate Sources and Sinks of OBD-II API
Annotate a source as Tainted
Parsed Bytecode
Information Flow Analysis for Taintedness
Access JVM Stack
Sink is Tainted?
Yes
Process and Throw Warning
NO
All Annotated Sources are Checked?
Yes
END

**FIGURE 10** Working principle of the OBD-II checker.

sink, annotated as @DeviceTrusted. Moreover, the value returned by such `get()` methods discloses sensitive information about the car. Consequently, it must be used in a proper way or otherwise privacy might be jeopardized. Hence, it is a source, annotated as @UntrustedDevice. Also, the parameter of the receive method of the listeners is a source, since it carries data reporting updates about the car status. Hence, it is annotated as @UntrustedDevice as well. Once such annotated interfaces are given to Julia, the analyzer can perform a taint analysis that is aware of those extra sources and sinks. Sources are marked as tainted during the analysis and then propagated. Sinks are checked for taintedness at the end of the analysis: if they are tainted, Julia issues a warning about a potential injection into the CAN device.

Listing 22: Java classes of OpenXC and their methods that allow Android apps to interact with the CAN.

```java
public class VehicleManager extends Service ... {

   // read a measurement from the CAN
   public @UntrustedDevice Measurement get(@DeviceTrusted Class<? extends Measurement> measurementType);

   // read a message from the CAN
   public @UntrustedDevice VehicleMessage get(@DeviceTrusted MessageKey key);

   // read a message from the CAN, waiting up to 2 seconds
   public @UntrustedDevice VehicleMessage request(@DeviceTrusted KeyedMessage message);

   // set a measurement to the CAN
   public boolean send(@DeviceTrusted Measurement message);

   // send a message to the CAN
   public boolean send(@DeviceTrusted VehicleMessage message);

   // send a simple message to the CAN and yields the result from the CAN
   public String requestCommandMessage(@DeviceTrusted CommandType type);

   // register a listener for receiving updates to the given message
   public void request(@DeviceTrusted KeyedMessage message, VehicleMessage.Listener listener);

   // register a listener for receiving updates to the given measurement
   public void addListener(@DeviceTrusted Class<? extends Measurement> measurementType, Measurement.Listener
        listener);
   // yield the device identifier of the vehicle interface
   public @UntrustedDevice String getVehicleInterfaceDeviceId();

   // yield the firmware version of the vehicle interface
   public @UntrustedDevice String getVehicleInterfaceVersion();
}

public interface Measurement {
   public interface Listener {
      // get notified about a measurement change from the CAN
      public void receive(@UntrustedDevice Measurement measurement);
   }
}

public class VehicleMessage ... {
   public interface Listener {
      // get notified about a new received message from the CAN
      public void receive(@UntrustedDevice VehicleMessage message);
   }
}

public class UserSink {
   // get notified about a measurement change from the CAN
   public void receive(@UntrustedDevice VehicleMessage measurement);
}

public class ApplicationSource {
   // get notified about a new received message from the CAN
   void handleMessage(@UntrustedDevice VehicleMessage message);
}

public class UsbVehicleInterface {
   // send raw data to the CAN through the USB interface
   boolean write(@DeviceTrusted byte[] bytes);
}

public class NetworkVehicleInterface {
   // send raw data to the CAN through the network interface
   boolean write(@DeviceTrusted byte[] bytes);
}

public class BluetoothVehicleInterface {
   // send raw data to the CAN through the Bluetooth interface
   boolean write(@DeviceTrusted byte[] bytes);
}
```

2. *Taint analysis*: The OBD-II analyzer builds on top of Julia. Among its checkers, Julia includes the Injection checker that implements a sound information flow analysis[26]. It propagates tainted data along all possible information flows. Boolean variables stand for program variables. Boolean formulas model explicit information flows. Namely, their models form a sound overapproximation of all taintedness behaviors for the variables in scope at a given program point. For instance, the abstraction of the `load k` bytecode instruction, that pushes on the operand stack the value of local variable k, is the Boolean formula $(\check{l}_k \leftrightarrow \hat{s}_{top}) \wedge U$, stating that the taintedness of the topmost stack element after this instruction ($\hat{s}_{top}$) is equal to the taintedness of local variable $k$ before the instruction ($\check{l}_k$); all other local variables and stack elements do not change (expressed by a formula $U$); taintedness before and after an instruction is distinguished by using distinct hats for the variables. There are such formulas for each bytecode instruction. Instructions that might have side-effects (field updates, array writes and method calls) need some approximation of the heap, to model the possible effects of the updates. The analysis of sequential instructions is merged through a sequential composition of formulas. Loops and recursion are saturated by fixpoint. The resulting analysis is a denotational, bottom-up taint analysis, that Julia implements through efficient binary decision diagrams[46].

## 6 | EXPERIMENTAL RESULTS AND DISCUSSION

This section presents the experiments of analysis with our infotainment and OBD-II checker, discussing the results. They have been performed on a desktop Intel Core i7 machine with 16GB of RAM.

## 6.1 | Analysis with the Infotainment Checker

We have analyzed Android Auto infotainment apps with the analyzer from Section 5.2.2. To this goal, Android Auto apps have been collected from the Google Play store. There are about 300 apps in the Android Auto infotainment category (as of July 2018). Hence, we have only selected those that have at least half a million installations, and whose rating is at least 4.0. This leads to 40 apps only (excluding WhatsApp, Google and Facebook apps, since they are highly obfuscated). These apps have been converted into Java bytecode with `dex2jar` and their manifest has been extracted through `apktool`. The resulting jar files, along with their manifest, have been fed into the Android Auto infotainment checker. Table 3 shows the service specific entry points for each app. The numbers in column a are the classes where event handling such as click, focus etc. are implemented. Similarly, the numbers in columns b, c, d, e, f, g and h are the classes where the functionalities related to services, broadcast receivers, content providers, media browsers, WebView services, file storage and text messaging are implemented. These cover almost all user interaction in the apps.

The classes selected as entry points are then used to build the call graph of the app. Table 4 shows the reachability within an app's code in terms of Lines Of Code (LOC), from the entry points. To compute the number of LOCs in an app, first the apps jar files are extracted using *dex2jar*[23], then we decompile these jar files using *Java Decompiler*[47] and compute the LOC. However, the decompiled code contains many precompiled libraries that are highly obfuscated and it is difficult to exclude them as we cannot retrieve the original package information. The majority of these libraries are put in parallel with the classes of the application and cannot be considered as dependency. While calculating the LOCs, these packages are exploded class by class hence increasing the LOC count. The results show that, with the selected entry points, we reached between 50% and 75% of the overall code. This moderate percentage is due to the fact that only classes reachable from the entry points have been considered. Furthermore, precompiled libraries have been ignored. Hence, the table shows that the selected entry points cover the majority of the apps' code, where their logic is implemented. This avoids checking every class in an app, hence reducing the analysis time.

Table 5 reports the vulnerabilities found in these apps and Figure 11 plots them in a graph. The numbers indicate how many warnings have been issued for a specific kind of vulnerability. The analysis detected vulnerabilities related to JavaScript execution and file and cache directory access in almost 60% of the apps; out of these, 25% show JavaScript execution threats. Furthermore, almost all apps use a large number of unprotected intents. We will show later that some of these unprotected intents can be easily exploited by an intruder to perform some attack. The analysis shows that 12% of the apps use weak encryption techniques, where the encryption key can be easily broken. However, none of these apps communicates with the car's internal ECUs. The analysis shows that apps do not leak the GPS location and all apps support voice operation. Moreover, these apps do not display any media advertisement, which would violate Google's requirements, and they do not use media autoplay. The fact that the analysis found only a few kinds of vulnerabilities is explained since most Android Auto apps use Google Now services for location, event processing, voice input, notification services etc. thus securing the apps interaction with the driver.

To check the severity of the warnings, we looked into the decompiled source code. For this purpose, the podcastaddict app was chosen, as it produced 11 warnings. For an example, the com.bamnetworks.mobile.android.gameday.activities.BlackoutActivity class is considered, that is accessible since it is an activity. It calls both the setJavaScriptEnabled() and the addJavaScriptInterface() methods. The code snippet in Listing 23 shows that the first argument to addJavaScriptInterface() is not a constant. Instead, it is initialized multiple times, even once with an intent call. Furthermore, method setJavaScriptEnabled() receives true as argument, which allows execution of JavaScript.

Listing 23: The argument passed to addJavaScriptInterface() is not a constant.

```java
package com.bamnetworks.mobile.android.gameday.activities;
public class BlackoutActivity extends AtBatDrawerActivity {
   public void onCreate(Bundle paramBundle) {
      ...
      paramBundle = getIntent().getStringExtra("zip");
      ...
      paramBundle = ((Geocoder)localObject).getFromLocationName(prmBundle + " " + str, 1);
      ...
      paramBundle = new BlackoutActivity.BlackoutMapJavaScriptInterface(this, d1, d2);
      this.blackoutMap.addJavaScriptInterface(paramBundle, "android");
      this.blackoutMap.getSettings().setJavaScriptEnabled(true);
      ...
   }
}
```

**TABLE 3** Entry points for the infotainment apps.

| # | App Name | a | b | c | d | e | f | g | h | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | smartaudiobookplayer (3.3.5) | 32 | 8 | 10 | 0 | 2 | 1 | 0 | 1 | 54 |
| 2 | abcnews (3.3.5) | 88 | 68 | 28 | 4 | 0 | 23 | 1 | 2 | 214 |
| 3 | itunerfree (4.2.10) | 50 | 20 | 24 | 0 | 0 | 23 | 3 | 3 | 123 |
| 4 | audible (2.12.0) | 136 | 68 | 36 | 16 | 3 | 4 | 0 | 2 | 265 |
| 5 | audiobooks (4.64) | 12 | 6 | 10 | 0 | 1 | 1 | 0 | 1 | 31 |
| 6 | podcastaddict (3.43.8) | 128 | 30 | 34 | 6 | 4 | 10 | 0 | 0 | 212 |
| 7 | MLB.com At Bat (5.6.0) | 148 | 28 | 18 | 2 | 3 | 20 | 3 | 2 | 224 |
| 8 | textplus (7.0.7) | 184 | 40 | 30 | 10 | 0 | 32 | 0 | 4 | 300 |
| 9 | icq mobile (6.13) | 86 | 62 | 48 | 6 |  | 2 | 0 | 5 | 209 |
| 10 | itunestoppodcastplayer (2.8.10) | 52 | 20 | 14 | 0 | 0 | 0 | 0 | 1 | 87 |
| 11 | jetaudio (8.2.3) | 88 | 8 | 24 | 0 | 1 | 3 | 0 | 2 | 126 |
| 12 | overdrive (3.6.2) | 28 | 16 | 2 | 8 | 1 | 4 | 0 | 2 | 61 |
| 13 | spotify (8.4.11.1283) | 240 | 112 | 36 | 8 | 0 | 8 | 25 | 5 | 434 |
| 14 | stitcher radio (3.9.8) | 88 | 26 | 24 | 2 | 0 | 9 | 0 | 0 | 149 |
| 15 | simpleradio (2.2.5.1) | 24 | 14 | 14 | 4 | 1 | 6 | 0 | 0 | 63 |
| 16 | deezer (5.4.8.46) | 162 | 36 | 30 | 12 | 0 | 23 | 21 | 2 | 286 |
| 17 | fm player (3.7.4.0) | 70 | 38 | 24 | 6 | 0 | 1 | 0 | 5 | 144 |
| 18 | kik (11.29.0.17461) | 40 | 280 | 10 | 8 | 0 | 8 | 12 | 0 | 358 |
| 19 | beyondpod (4.2.16) | 62 | 12 | 18 | 4 | 0 | 6 | 0 | 2 | 104 |
| 20 | npr (1.7.2.2) | 44 | 42 | 14 | 6 | 2 | 1 | 0 | 1 | 110 |
| 21 | tunein player (18.3.1) | 94 | 32 | 44 | 20 | 0 | 13 | 0 | 3 | 206 |
| 22 | sevendigital (6.69.226) | 66 | 86 | 22 | 2 | 0 | 27 | 0 | 3 | 206 |
| 23 | Librivox (7.4.2) | 21 | 10 | 6 | 3 | 0 | 1 | 0 | 1 | 42 |
| 24 | myTunerFree (5.2.5) | 64 | 14 | 12 | 3 | 0 | 19 | 0 | 1 | 113 |
| 25 | Cisco Webex Meetings (10.0.0) | 31 | 15 | 11 | 1 | 0 | 3 | 0 | 1 | 62 |
| 26 | Disa (0.9.9.3) | 34 | 5 | 23 | 3 | 0 | 1 | 3 | 0 | 69 |
| 27 | doubleTwist (3.1.6) | 7 | 2 | 2 | 0 | 0 | 1 | 0 | 0 | 12 |
| 28 | Rocket Music Player (5.2) | 32 | 17 | 16 | 4 | 0 | 3 | 3 | 0 | 75 |
| 29 | Quran Pro (1.7.67) | 21 | 15 | 11 | 8 | 0 | 2 | 0 | 4 | 61 |
| 30 | Waze (4.37.0.6) | 205 | 14 | 13 | 0 | 0 | 2 | 0 | 2 | 236 |
| 31 | Cherie (4.1.14) | 30 | 18 | 15 | 1 | 0 | 11 | 0 | 1 | 76 |
| 32 | NRJ Radio (4.4.13) | 30 | 18 | 15 | 1 | 0 | 15 | 0 | 3 | 82 |
| 33 | GoneMAD Music Player (2.0.28) | 16 | 6 | 9 | 0 | 0 | 1 | 0 | 1 | 33 |
| 34 | Radio Deejay (3.3.2) | 29 | 7 | 8 | 1 | 0 | 7 | 0 | 1 | 53 |
| 35 | Les Indes Radios (5.0.2) | 8 | 7 | 7 | 1 | 1 | 0 | 10 | 0 | 34 |
| 36 | ZapZap (71.12) | 17 | 21 | 18 | 2 | 0 | 6 | 0 | 5 | 69 |
| 37 | Telegram (4.8.7) | 78 | 34 | 24 | 3 | 0 | 6 | 12 | 1 | 158 |
| 38 | Signal (4.17.5) | 42 | 9 | 16 | 4 | 0 | 0 | 24 | 0 | 95 |
| 39 | Agent (6.11) | 42 | 31 | 23 | 3 | 0 | 2 | 0 | 5 | 106 |
| 40 | Pulse SMS (2.11.1.2072) | 25 | 37 | 14 | 3 | 0 | 0 | 0 | 2 | 81 |

a) Activities. b) Services. c) Broadcast receivers. d) Content providers.
e) Media browser. f) WebView service. g) File storage. h) Text messaging.

**TABLE 4** Reachability analysis of infotainment apps.

| # | App Name | Entry points | Reachable LOCs | Total LOCs | Coverage(%) |
|---|---|---|---|---|---|
| 1 | smartaudiobookplayer (3.3.5) | 54 | 76426 | 109902 | 69.54 |
| 2 | abcnews (3.3.5) | 214 | 405769 | 659941 | 61.49 |
| 3 | itunerfree (4.2.10) | 123 | 275225 | 451724 | 60.93 |
| 4 | audible (2.12.0) | 265 | 320497 | 452634 | 70.81 |
| 5 | audiobooks (4.64) | 31 | 124551 | 182160 | 68.37 |
| 6 | podcastaddict (3.43.8) | 212 | 194195 | 353620 | 54.92 |
| 7 | MLB.com At Bat (5.6.0) | 224 | 297209 | 591157 | 50.28 |
| 8 | textplus (7.0.7) | 300 | 338302 | 586680 | 57.66 |
| 9 | icq mobile (6.13) | 209 | 209691 | 322360 | 65.05 |
| 10 | itunestoppodcastplayer (2.8.10) | 87 | 178836 | 324181 | 55.17 |
| 11 | jetaudio (8.2.3) | 126 | 83295 | 115714 | 71.98 |
| 12 | overdrive (3.6.2) | 61 | 172651 | 336762 | 51.27 |
| 13 | spotify (8.4.11.1283) | 434 | 370505 | 773512 | 47.90 |
| 14 | stitcher radio (3.9.8) | 149 | 239605 | 389262 | 61.55 |
| 15 | simpleradio (2.2.5.1) | 63 | 189583 | 341488 | 55.52 |
| 16 | deezer (5.4.8.46) | 286 | 400704 | 791891 | 50.60 |
| 17 | fm player (3.7.4.0) | 144 | 173005 | 311657 | 55.51 |
| 18 | kik (11.29.0.17461) | 358 | 359798 | 492574 | 73.04 |
| 19 | beyondpod (4.2.16) | 104 | 130085 | 271281 | 47.95 |
| 20 | npr (1.7.2.2) | 110 | 214492 | 361362 | 59.36 |
| 21 | tunein player (18.3.1) | 206 | 281758 | 438734 | 64.22 |
| 22 | sevendigital (6.69.226) | 206 | 214219 | 375949 | 56.98 |
| 23 | Librivox (7.4.2) | 42 | 84705 | 101695 | 83.29 |
| 24 | myTunerFree (5.2.5) | 113 | 123518 | 225516 | 54.77 |
| 25 | Cisco Webex Meetings (10.0.0) | 62 | 170204 | 279345 | 60.93 |
| 26 | Disa (0.9.9.3) | 69 | 110803 | 163461 | 67.79 |
| 27 | doubleTwist (3.1.6) | 12 | 130382 | 218944 | 59.55 |
| 28 | Rocket Music Player (5.2) | 75 | 79248 | 151560 | 52.29 |
| 29 | Quran Pro (1.7.67) | 61 | 65031 | 113549 | 57.27 |
| 30 | Waze (4.37.0.6) | 236 | 123896 | 253869 | 48.80 |
| 31 | Cherie (4.1.14) | 76 | 152089 | 290757 | 52.31 |
| 32 | NRJ Radio (4.4.13) | 82 | 162885 | 290754 | 56.02 |
| 33 | GoneMAD Music Player (2.0.28) | 33 | 63813 | 73767 | 86.51 |
| 34 | Radio Deejay (3.3.2) | 53 | 84160 | 203931 | 41.27 |
| 35 | Les Indes Radios (5.0.2) | 34 | 136495 | 195995 | 69.64 |
| 36 | ZapZap (71.12) | 69 | 249445 | 501715 | 49.72 |
| 37 | Telegram (4.8.7) | 158 | 159895 | 202378 | 79.01 |
| 38 | Signal (4.17.5) | 95 | 81385 | 187987 | 43.29 |
| 39 | Agent (6.11) | 106 | 139684 | 194705 | 71.74 |
| 40 | Pulse SMS (2.11.1.2072) | 81 | 69232 | 126259 | 54.83 |

Julia reports an external file access warning in this app. The code snippet in Listing 24 shows that the app saves data in an external device, that can be deleted by the system, the user or other app to save space, hence leaving the app in a vulnerable state.

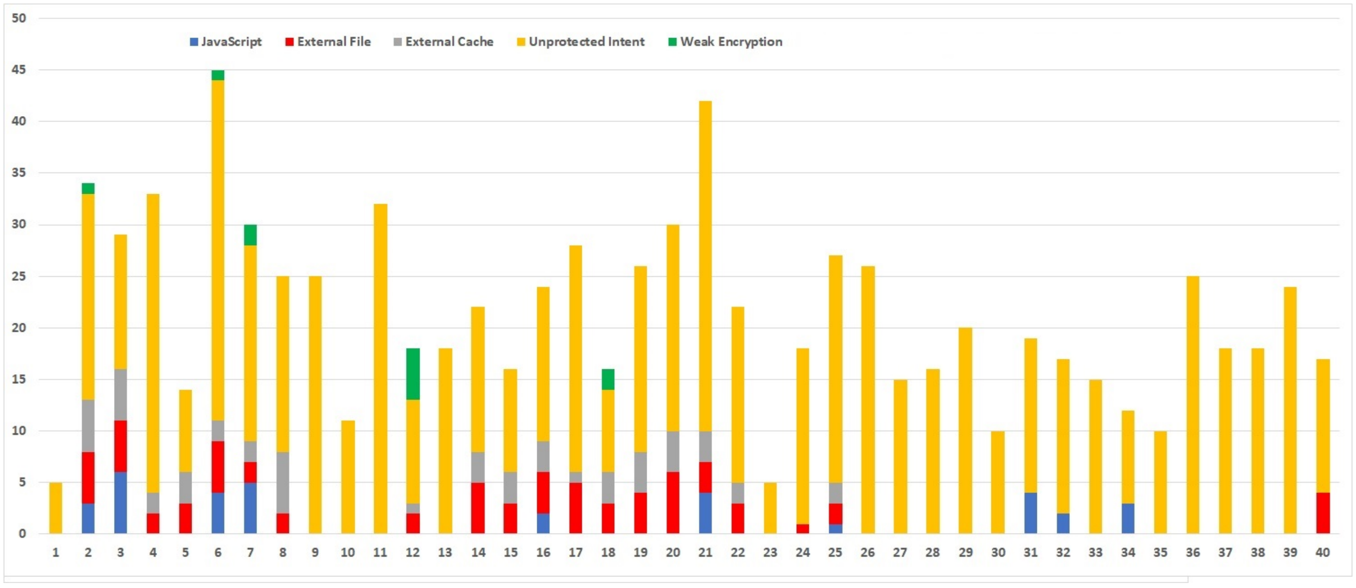


**FIGURE 11** Number of warnings in the apps.

Listing 24: App data stored in an external device.

```
package com.bambuna.podcastaddict.h;
public static List<String> a(Context paramContext) {
  ...
  ArrayList localArrayList = new ArrayList();
  if (paramContext != null) {
    paramContext = a.getExternalFilesDirs(paramContext, null);
    ...
  }
}
```

Let us consider the unprotected intents warnings from Table 5 . In general, these are related to exported content providers. As the detected intents are not protected by permissions, they are more vulnerable to attacks. To check their exploitability, we have considered the *null fuzzing*[12] of intents in the apps. In general, intents facilitate inter- and intra-app communication among components (activities, services, broadcast receivers, content providers). Intent fuzzing consists in passing carefully crafted intents to the app in order to discover the vulnerability in the communication between components. However, due to the large number of apps that we are testing, we have considered fuzzing apps with null intents only, that is, by injecting intents with blank data. The test coverage of this approach is quite restricted but it successfully demonstrates the exploitability of these unprotected intents with a simple fuzzing attack. Interestingly, this allows one to start many unprotected activities of the apps. Malicious apps can perform an escalation attack stealthily, provided the user performs the right interaction that triggers the harmful action, which limits the severity of the attack. Furthermore, in two apps, service intents let one access the notifications. Namely, intent msa.apps.podcastplayer.player.PlaybackService of the iTunePodcastPlayerv app and intent com.appgeneration.ituner.auto.AutoMediaBrowserService of the myTunerFree app generate abrupt notifications when crafted with null intents. Interestingly, null fuzzing of these service intents and of some activity intents leaves the apps in a non-responsive state, which leads to disruption of other important functionality of the infotainment system. Furthermore, the experimental results show that, in all apps, there are some unprotected intents that provide direct access to the targeted app components, shown as Reactive in Figure 12 ; some of them even bypass the authentication process. Moreover, some unprotected intents receive the crafted null intents but do not provide access to the targeted components, shown as Non Reactive in Figure 12 . These intents are vulnerable in the sense that one can trigger these components by injecting crafted intents with suitable data, instead of null intents. However, a few unprotected intents cannot be fuzzed from the outside, shown as Not Accessible in Figure 12 .

[12]Injecting intents with blank data.

**TABLE 5** Result of the Android Auto checker of Julia.

| # | App Name | Java Script | External File | External Cache | Unprotected Intent | Weak Encryption | Total |
|---|---|---|---|---|---|---|---|
| 1 | smartaudiobookplayer (3.3.5) | 0 | 0 | 0 | 5 | 0 | 5 |
| 2 | abcnews (3.3.5) | 3 | 5 | 5 | 20 | 1 | 34 |
| 3 | itunerfree (4.2.10) | 6 | 5 | 5 | 13 | 0 | 29 |
| 4 | audible (2.12.0) | 0 | 2 | 2 | 29 | 0 | 33 |
| 5 | audiobooks (4.64) | 0 | 3 | 3 | 8 | 0 | 14 |
| 6 | podcastaddict (3.43.8) | 4 | 5 | 2 | 33 | 1 | 45 |
| 7 | MLB.com At Bat (5.6.0) | 5 | 2 | 2 | 19 | 2 | 30 |
| 8 | textplus (7.0.7) | 0 | 2 | 6 | 17 | 0 | 25 |
| 9 | icq mobile (6.13) | 0 | 0 | 0 | 25 | 0 | 25 |
| 10 | itunestoppodcastplayer (2.8.10) | 0 | 0 | 0 | 11 | 0 | 11 |
| 11 | jetaudio (8.2.3) | 0 | 0 | 0 | 32 | 0 | 32 |
| 12 | overdrive (3.6.2) | 0 | 2 | 1 | 10 | 5 | 18 |
| 13 | spotify (8.4.11.1283) | 0 | 0 | 0 | 18 | 0 | 18 |
| 14 | stitcher radio (3.9.8) | 0 | 5 | 3 | 14 | 0 | 22 |
| 15 | simpleradio (2.2.5.1) | 0 | 3 | 3 | 10 | 0 | 16 |
| 16 | deezer (5.4.8.46) | 2 | 4 | 3 | 15 | 0 | 24 |
| 17 | fm player (3.7.4.0) | 0 | 5 | 1 | 22 | 0 | 28 |
| 18 | kik (11.29.0.17461) | 0 | 3 | 3 | 8 | 2 | 16 |
| 19 | beyondpod (4.2.16) | 0 | 4 | 4 | 18 | 0 | 26 |
| 20 | npr (1.7.2.2) | 0 | 6 | 4 | 20 | 0 | 30 |
| 21 | tunein player (18.3.1) | 4 | 3 | 3 | 32 | 0 | 42 |
| 22 | sevendigital (6.69.226) | 0 | 3 | 2 | 17 | 0 | 22 |
| 23 | Librivox (7.4.2) | 0 | 0 | 0 | 5 | 0 | 5 |
| 24 | myTunerFree (5.2.5) | 0 | 1 | 0 | 17 | 0 | 18 |
| 25 | Cisco Webex Meetings (10.0.0) | 1 | 2 | 2 | 22 | 0 | 27 |
| 26 | Disa (0.9.9.3) | 0 | 0 | 0 | 26 | 0 | 26 |
| 27 | doubleTwist (3.1.6) | 0 | 0 | 0 | 15 | 0 | 15 |
| 28 | Rocket Music Player (5.2) | 0 | 0 | 0 | 16 | 0 | 16 |
| 29 | Quran Pro (1.7.67) | 0 | 0 | 0 | 20 | 0 | 20 |
| 30 | Waze (4.37.0.6) | 0 | 0 | 0 | 10 | 0 | 10 |
| 31 | Cherie (4.1.14) | 4 | 0 | 0 | 15 | 0 | 19 |
| 32 | NRJ Radio (4.4.13) | 2 | 0 | 0 | 15 | 0 | 17 |
| 33 | GoneMAD Music Player (2.0.28) | 0 | 0 | 0 | 15 | 0 | 15 |
| 34 | Radio Deejay (3.3.2) | 3 | 0 | 0 | 9 | 0 | 12 |
| 35 | Les Indes Radios (5.0.2) | 0 | 0 | 0 | 10 | 0 | 10 |
| 36 | ZapZap (71.12) | 0 | 0 | 0 | 25 | 0 | 25 |
| 37 | Telegram (4.8.7) | 0 | 0 | 0 | 18 | 0 | 18 |
| 38 | Signal (4.17.5) | 0 | 0 | 0 | 18 | 0 | 18 |
| 39 | Agent (6.11) | 0 | 0 | 0 | 24 | 0 | 24 |
| 40 | Pulse SMS (2.11.1.2072) | 0 | 4 | 0 | 13 | 0 | 17 |

For comparison, we have run the QARK analyzer on the same apps. QARK is a comprehensive static analyzer for Android apps, that looks for a wide range of standard smartphone vulnerabilities, such as WebViews, file permissions and cryptography. The results are in Table 6 . QARK is not designed for infotainment apps, hence it does not provide checks specific to such apps.

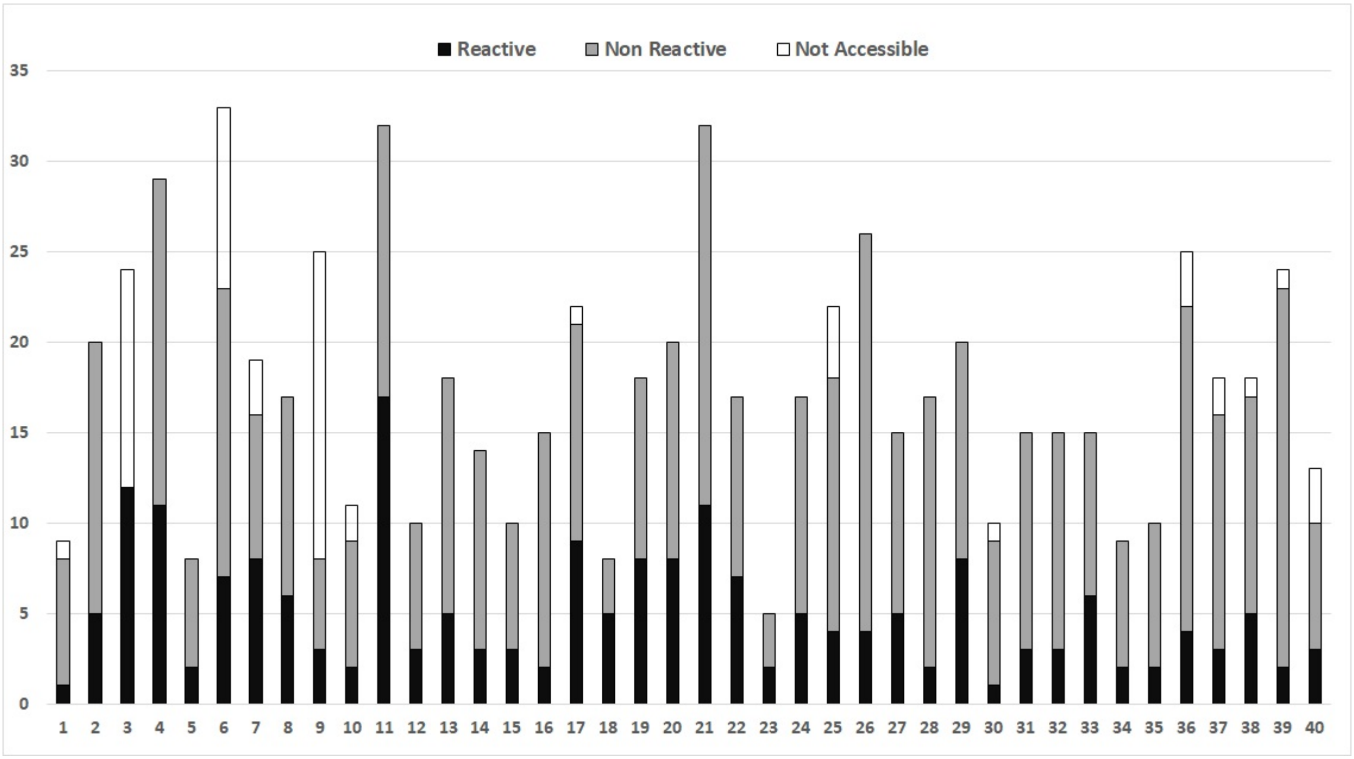


**FIGURE 12** Null fuzzing of unprotected intents.

Moreover, its analysis crashed on many apps. Figure 13 shows a comparison between the results of Julia and those of QARK. This table restricts the results of Julia to only those warnings that correspond to issues covered by QARK as well. It can be seen that QARK hung or crashed for the analysis of 9 apps over 40. Moreover, both analyzers issue a similar number of warnings for encryption and intents. However, in most cases QARK does not detect issues related to JavaScript execution and external file or external cache access, such as those in the MLB.com at BAT app, reported in the code snippet of Listings 23 and 24. These results show that our infotainment checker provides a more reliable and more precise static analysis solution than QARK.

## 6.2 | Analysis with the OBD-II Checker

We have analyzed the OpenXC apps from Section 4.2 with Julia's OBD-II checker, instantiated with the annotations in Listing 22. Tables 7 and Table 8 show entry points and percentage of reachable code, similarly to the Infotainment checker. These analyses required up to 3 minutes per app.

In the Rain Monitor app, the taint analysis issues the five warnings about potential injections reported in Listing 25. They correspond to the injections informally discussed in Item 1 of Section 2.2. In particular, the first warning of Listing 25 corresponds to that discussed in the code snippet in Listing 15: sensitive data about the car flows into the `execute` method that builds an HTTP request. This is definitely a dangerous injection, although not exactly a cross-site scripting (XSS) injection, as the analyzer suggests, since it cannot distinguish the source of tainted data. Moreover, the status of the HTTP request and the status of the windshield get logged into a file, as shown in the code snippet of Listing 15 (second and third warning of Listing 25). The former is an example of data coming from the external world (the HTTP server might be compromised and send any possible status); the latter is an example of sensitive data about the car. In the code snippet of Listing 16, sensitive data (latitude and longitude) is read from the CAN, logged at line 68 (fourth warning of Listing 25) and later used to build a URL at line 76 (fifth warning of Listing 25). The latter points to a remote web service that tracks the position of the car and the weather. Clearly, this is potentially a privacy breach. In conclusion, the OBD-II checker of Julia issues five injection warnings on Rain Monitor and they are all true alarms, although inherent to the task the app has to perform.

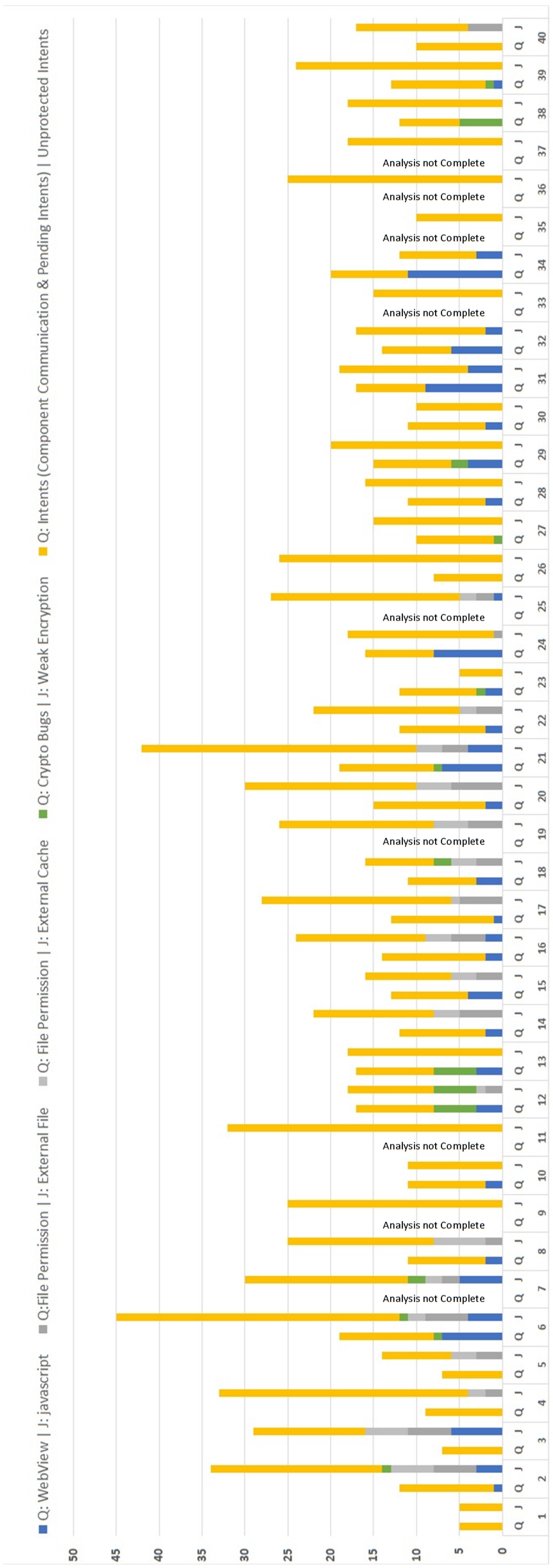


**FIGURE 13** Comparative analysis of results with Julia and QARK.

**TABLE 6** Analysis of Android Auto infotainment apps with QARK.

| Sl No | App Name | Web View | File Permission | Crypto Bugs | Intents |  | Total |
|---|---|---|---|---|---|---|---|
|  |  |  |  |  | Component Communication | Pending Intent |  |
| 1 | smartaudiobookplayer (3.3.5) | 0 | 0 | 0 | 3 | 2 | 5 |
| 2 | abcnews (3.3.5) | 1 | 0 | 0 | 4 | 7 | 12 |
| 3 | itunerfree (4.2.10) | 0 | 0 | 0 | 4 | 3 | 7 |
| 4 | audible (2.12.0) | 0 | 0 | 0 | 5 | 4 | 9 |
| 5 | audiobooks (4.64) | 0 | 0 | 0 | 4 | 3 | 7 |
| 6 | podcastaddict (3.43.8) | 7 | 0 | 1 | 6 | 5 | 19 |
| 7 | MLB.com At Bat (5.6.0) | Analysis Not Complete |  |  |  |  |  |
| 8 | textplus (7.0.7) | 2 | 0 | 0 | 4 | 5 | 11 |
| 9 | icq mobile (6.13) | Analysis Not Complete |  |  |  |  |  |
| 10 | itunestoppodcastplayer (2.8.10) | 2 | 0 | 0 | 3 | 6 | 11 |
| 11 | jetaudio (8.2.3) | Analysis Not Complete |  |  |  |  |  |
| 12 | overdrive (3.6.2) | 3 | 0 | 5 | 5 | 4 | 17 |
| 13 | spotify (8.4.11.1283) | 3 |  | 5 | 5 | 4 | 17 |
| 14 | stitcher radio (3.9.8) | 2 | 0 | 0 | 5 | 5 | 12 |
| 15 | simpleradio (2.2.5.1) | 4 | 0 | 0 | 4 | 5 | 13 |
| 16 | deezer (5.4.8.46) | 2 | 0 | 0 | 5 | 7 | 14 |
| 17 | fm player (3.7.4.0) | 1 | 0 | 0 | 5 | 7 | 13 |
| 18 | kik (11.29.0.17461) | 3 | 0 | 0 | 5 | 3 | 11 |
| 19 | beyondpod (4.2.16) | Analysis Not Complete |  |  |  |  |  |
| 20 | npr (1.7.2.2) | 2 | 0 | 0 | 5 | 8 | 15 |
| 21 | tunein player (18.3.1) | 7 | 0 | 1 | 6 | 5 | 19 |
| 22 | sevendigital (6.69.226) | 2 | 0 | 0 | 5 | 5 | 12 |
| 23 | Librivox (7.4.2) | 2 | 0 | 1 | 6 | 3 | 12 |
| 24 | myTunerFree (5.2.5) | 8 | 0 | 0 | 4 | 4 | 16 |
| 25 | Cisco Webex Meetings (10.0.0) | Analysis Not Complete |  |  |  |  |  |
| 26 | Disa (0.9.9.3) | 0 | 0 | 0 | 4 | 4 | 8 |
| 27 | doubleTwist (3.1.6) | 0 | 0 | 1 | 6 | 3 | 10 |
| 28 | Rocket Music Player (5.2) | 2 | 0 | 0 | 5 | 4 | 11 |
| 29 | Quran Pro (1.7.67) | 4 | 0 | 2 | 4 | 5 | 15 |
| 30 | Waze (4.37.0.6) | 2 | 0 | 0 | 5 | 4 | 11 |
| 31 | Cherie (4.1.14) | 9 | 0 | 0 | 4 | 4 | 17 |
| 32 | NRJ Radio (4.4.13) | 6 | 0 | 0 | 4 | 4 | 14 |
| 33 | GoneMAD Music Player (2.0.28) | Analysis Not Complete |  |  |  |  |  |
| 34 | Radio Deejay (3.3.2) | 11 | 0 | 0 | 5 | 4 | 20 |
| 35 | Les Indes Radios (5.0.2) | Analysis Not Complete |  |  |  |  |  |
| 36 | ZapZap (71.12) | Analysis Not Complete |  |  |  |  |  |
| 37 | Telegram (4.8.7) | Analysis Not Complete |  |  |  |  |  |
| 38 | Signal (4.17.5) | 0 | 0 | 5 | 5 | 2 | 12 |
| 39 | Agent (6.11) | 1 | 0 | 1 | 6 | 5 | 13 |
| 40 | Pulse SMS (2.11.1.2072) | 0 | 0 | 0 | 3 | 7 | 10 |

Listing 25: Vulnerability warnings for the Rain Monitor app.

```
CheckWipersTask.java:111:XSS-injection into method "execute"
CheckWipersTask.java:114:Log forging into method "w"
CheckWipersTask.java:117:Log forging into method "d"
FetchAlertsTak.java:68:Log forging into method "d"
FetchAlertsTak.java:76:URL injection into method "<init>"
```

**TABLE 7** Entry points for the OBD apps.

| # | App Name | a | b | c | d | e | f | g | h | i | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Connected Wiper | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 22 |
| 2 | Diagonstic App | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 37 | 43 |
| 3 | Enabler App | 5 | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 71 | 79 |
| 4 | MPG App | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 94 | 98 |
| 5 | NightVision App | 1 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 12 | 17 |
| 6 | Rain Monitor | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 11 |
| 7 | Shiftknob App | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 29 | 32 |
| 8 | PixelOpenXC App | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 69 | 72 |
| 9 | Signal monitor app | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 13 |
| 10 | Validation App | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 57 | 59 |

a) Activities. b) Services. c) Broadcast receivers. d) Content providers. e) Media browser. f) WebView service. g) File storage. h) Text messaging. i) Sources of tainted data.

**TABLE 8** Reachability analysis of the OBD apps.

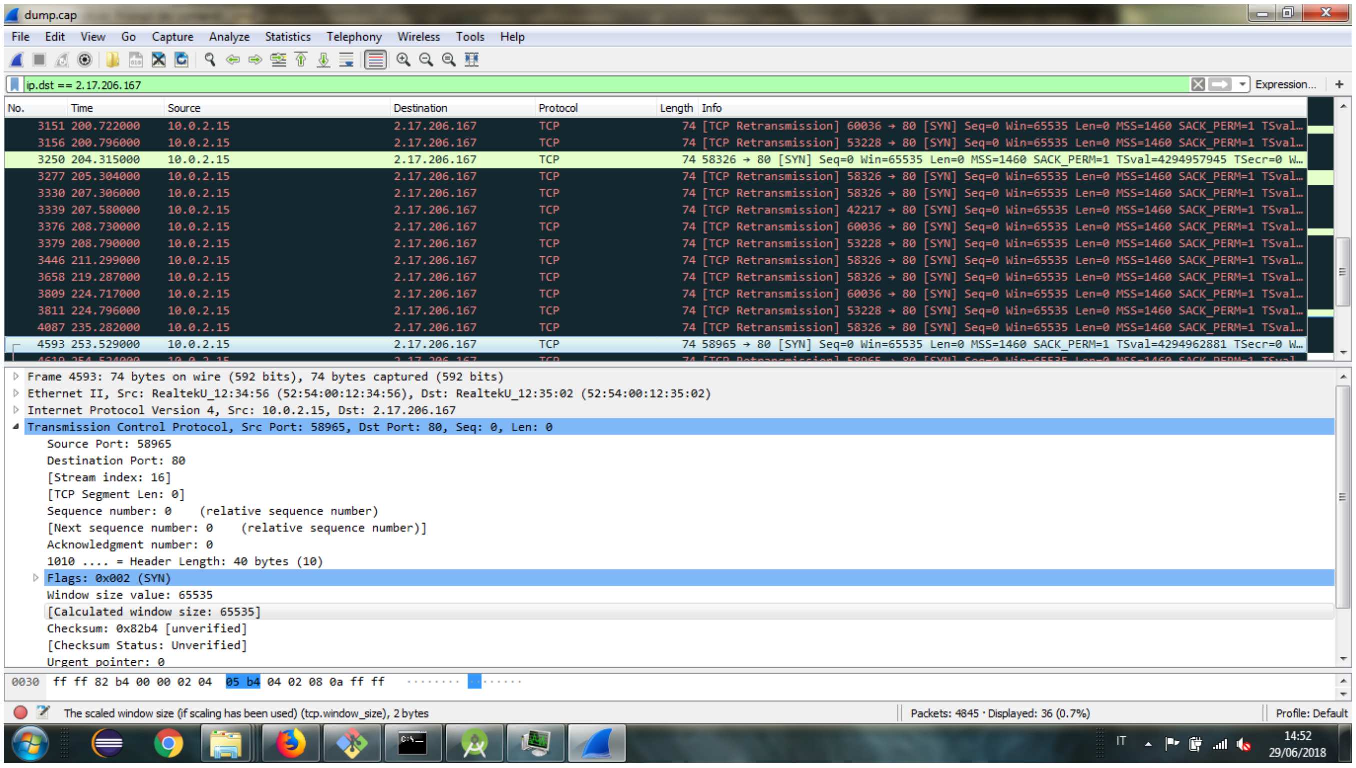
| # | App Name | Entry points | Reachable LOCs | Total LOCs | Coverage(%) |
|---|---|---|---|---|---|
| 1 | Connected Wiper | 22 | 103362 | 101606 | 93.45 |
| 2 | Diagonstic App | 0 | 4029 | 7809 | 51.59 |
| 3 | Enabler App | 79 | 12977 | 28190 | 46.03 |
| 4 | MPG App | 100 | 139308 | 203350 | 68.50 |
| 5 | NightVision App | 17 | 224 | 225 | 99.56 |
| 6 | Rain Monitor | 11 | 110 | 219 | 50.23 |
| 7 | Shiftknob App | 32 | 761 | 1969 | 38.65 |
| 8 | PixelOpenXC App | 72 | 10485 | 29861 | 35.11 |
| 9 | Signal monitor app | 13 | 286 | 894 | 31.99 |
| 10 | Validation App | 59 | 9157 | 10904 | 83.97 |

To check the correctness of the privacy issue warning (fifth warning in Listing 25) we have tested the packet transmission between the app and the web with WireShark. The results show that the app periodically sends data to the IP address *2.17.206.167* (Figure 14 ). Therefore the privacy issue detected at line 76 is a true positive.

Listing 26: Two vulnerability warnings for the OpenXC Enabler app.

```
SendCanMessageFragment.java:110:Device injection into method "send"
NetworkPreferenceManager.java:53:Log forging into method "w"
```

In the OpenXC Enabler app, Julia's OBD-II checker issues seven warnings about potential injections, including the two shown in the code snippet in Listing 26. The first corresponds to the injection discussed in Item 2 of Section 4.2 for the same app. Namely, data coming from user-controlled widgets flows into the send() method and hence to the CAN (Listing 17). The second warning in Listing 26 corresponds to the other discussed in the same section about the flow of user-controlled preferences into the logs (Listing 18). There is also a third warning, similar to the first one (line 127 of DiagnosticRequestFragment.java). Four more warnings are similar to the second one, that is, they warn about data coming from the preferences of the app (hence under user control) that can flow into logs (line 480 of SettingsActivity.java, line 69 of PreferenceManagerService.java and line 72 of TraceSourcePreferenceManager.java) or into the specification of a file name (*path-traversal*: line 72 of viewTraces.java). All seven warnings are true alarms.


dump.cap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression…

```
ip.dst == 2.17.206.167

No.   Time        Source      Destination   Protocol  Length Info
3151 200.722000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 60036 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3156 200.796000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 53228 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3250 204.315000  10.0.2.15   2.17.206.167  TCP       74 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=4294957945 TSecr=0 W…
3277 205.304000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3330 207.306000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3339 207.580000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 42217 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3376 208.730000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 60036 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3379 208.790000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 53228 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3446 211.299000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3658 219.287000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3809 224.717000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 60036 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
3811 224.796000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 53228 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
4087 235.282000  10.0.2.15   2.17.206.167  TCP       74 [TCP Retransmission] 58326 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval…
4593 253.529000  10.0.2.15   2.17.206.167  TCP       74 58965 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=4294962881 TSecr=0 W…

Frame 4593: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: RealtekU_12:34:56 (52:54:00:12:34:56), Dst: RealtekU_12:35:02 (52:54:00:12:35:02)
Internet Protocol Version 4, Src: 10.0.2.15, Dst: 2.17.206.167
Transmission Control Protocol, Src Port: 58965, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 58965
    Destination Port: 80
    [Stream index: 16]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    [Next sequence number: 0    (relative sequence number)]
    Acknowledgment number: 0
    1010 .... = Header Length: 40 bytes (10)
  Flags: 0x002 (SYN)
    Window size value: 65535
    [Calculated window size: 65535]
    Checksum: 0x82b4 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0

0030  ff ff 82 b4 00 00 02 04  05 b4 04 02 08 0a ff ff   ........ ......

The scaled window size (if scaling has been used) (tcp.window_size), 2 bytes    Packets: 4845 · Displayed: 36 (0.7%)    Profile: Default
IT  14:52 29/06/2018
```

**FIGURE 14** Package transmission detected by using WireShark.

For the apps Shift Knob and Night Vision discussed in Item 4 of Section 4.2, Julia issues no injection warnings. This is in line with the fact that sensitive data coming from the CAN flows in a controlled way inside those apps and is never used in a critical operation nor it is sent outside the device.

For the app MPG discussed in Item 3 of Section 4.2, Julia issues one injection warning at line 343 of MpgActivity.java. There, an integer option from Android preferences (hence controlled by the user) is used in a call to Thread.sleep(). This allows a denial-of-service injection by setting a large integer value in the preferences.

More interestingly, Julia does not issue any warning in the snippet of code in Listing 21. Method insertOrThrow() is not in the list of sinks provided to the analyzer, since it is known to sanitize data used to perform the SQL query. Hence, no warning is issued there.

These results of the OBD-II checker show the effectiveness of our analysis: it did not only identify several real issues, but also produced no false alarms (*noise*) in the analyzed apps.

The same apps have been analyzed with other static analysis tools (Table 9 ): FindBugs[13], SpotBugs[14], SonarQube[15] and QARK[16]. None of the above injections has been identified by these tools. Some of them do issue some warnings tagged as *security issues*, by using some syntactical check of the code. Namely, SonarQube complains about the fact that some public fields should have been declared as final, since they are never modified; or that some visibility modifier is too weak; it also complains about calls to File.delete() without checking the result value, which in Java is meant to inform about the outcome of the operation. Julia would issue the same warnings, had the corresponding checkers been turned on; however, it does not tag them as security issues but rather as bugs or inefficiencies. QARK issues warnings about a too small minSdkVersion in the AndroidManifest.xml, which is known to allow some security vulnerabilities; it also warns about the run-time registration of Android broadcast receivers, that might allow some form of data hijacking. None of them is currently considered by Julia. All these checks are simply syntactical checks of the code, in the sense that the analyzers do not make any effort in proving that the

[13]http://findbugs.sourceforge.net
[14]https://spotbugs.github.io
[15]https://www.sonarqube.org
[16]https://github.com/linkedin/qark

**TABLE 9** Analysis results for the OpenXC apps.

| CWE | Description | FindBugs | SpotBugs | SonarQube | QARK | **Julia** |
|---|---|---|---|---|---|---|
| 22 | Path Injection | 0 | 0 | 0 | 0 | **2** |
| 74 | Device Injection | 0 | 0 | 0 | 0 | **5** |
| 79 | XSS Injection | 0 | 0 | 0 | 0 | **1** |
| 117 | Log Forging | 0 | 0 | 0 | 0 | **39** |
| 501 | Trust Boundary Violation | 0 | 0 | 0 | 0 | **10** |
| 754 | Missing Check of Condition | 0 | 0 | 1 | 0 | **1** |
| 73 | External Control of File Name or Path | 0 | 0 | 0 | 1 | 0 |
| 264 | Permissions, Privileges | 0 | 0 | 0 | 10 | 0 |
| 374 | Passing Mutable Objects to an Untrusted Method | 1 | 1 | 0 | 0 | 0 |
| 476 | Null Pointer Dereference | 1 | 0 | 0 | 0 | 0 |
| 493 | Critical Public Field Without Final Modifier | 2 | 87 | 197 | 0 | 0 |
| 500 | Public Static Field Not Marked Final | 1 | 2 | 172 | 0 | 0 |
| 607 | Public Static Final Field References Mutable Object | 4 | 36 | 42 | 0 | 0 |
| 925 | Improper Check of Intent | 0 | 0 | 0 | 5 | 0 |
| | Total | 9 | 126 | 412 | 16 | **58** |

risk is real or only potential. They only match a code pattern. FindBugs and SpotBugs issue no security warnings on the five apps.

## 7 | CONCLUSION

As far as we know, this is the first static analysis for Android Auto infotainment and OBD-II apps, based on a formal basis such as abstract interpretation, that has been systematically applied to apps published in the Google Play store. The Auto infotainment checker only detects the vulnerabilities specified in Google's apps quality requirements. Whereas, the OBD-II analyzer is implemented for OpenXC API. The experimental results show that 60% of the apps expose some vulnerabilities, and 25% of them are related to JavaScript. While these types of vulnerabilities do not pose serious safety problems, they could affect the user experience (*e.g.*, by distracting the driver), and they could potentially lead to the rejection of the app from the Google Play store. Furthermore, most apps have a large number of unprotected intents and some of these intents can be easily exploited by an attacker. In addition, we also found out that 12% of the apps use weak encryption techniques, where the encryption key can be easily broken using existing techniques. Finally, the analysis results of the OpenXC-based OBD-II apps show possible severe CAN injections and privacy breaches that leak sensitive information to the web. These vulnerabilities might be effectively exploited by an attacker to compromise the privacy and safety of the car. Note that Julia does not automatically fix or propose fixes in order to address these issues (but it provides a detailed message about its findings), since they usually require the manual inspection of the code to understand where the issue is, if it is a true or a false alarm, and how to fix it while still preserving the functionality of the app.

Our future work includes the instantiation of the OBD-II checker to other OBD-II APIs. We are also contacting car manufacturers and companies developing Android Auto apps, in order to apply our analysis during the development of real-world apps.

**Acknowledgements**

*This work has been partially supported by Regione Veneto within the FSE project "Static analysis for the safety and security of Android systems for automotive infotainment" and by the CINI Cybersecurity National Laboratory within the project "FilieraSicura".*

## References

1. Budnick Noah. Largest Distracted Driving Behavior Study. http://blog.zendrive.com/distracted-driving/.Accessed On: 4-Jan-2019.
2. Apple . Apple CarPlay The ultimate copilot. https://www.apple.com/ios/carplay/.Accessed On: 18-Aug-2018.
3. Google . Android AUto. https://www.android.com/auto/.Accessed On: 18-Aug-2018.
4. Store Google Play. Apps for Android Auto. https://play.google.com/store/apps/collection/promotion_3001303_android_auto_all?hl=en.Accessed On: 18-Aug-2018.
5. Miller Charlie, Valasek Chris. Adventures in automotive networks and control units. *Def Con.* 2013;21:260–264.
6. Koscher Karl, Czeskis Alexei, Roesner Franziska, et al. Experimental security analysis of a modern automobile. *2010 IEEE Symposium on Security and Privacy (SP), Berkeley, CA, USA, May 22-25.* 2010;:447–462.
7. Nirumand Atefeh, Zamani Bahman, Tork Ladani Behrouz. VAnDroid: A framework for vulnerability analysis of Android applications using a model-driven reverse engineering technique. *Software: Practice and Experience.* ;0(0):1–30.
8. Checkoway Stephen, McCoy Damon, Kantor Brian, et al. Comprehensive experimental analyses of automotive attack surfaces. *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12.* 2011;:77–92.
9. Dardanelli Andrea, Maggi Federico, Tanelli Mara, et al. A security layer for smartphone-to-vehicle communication over bluetooth. *IEEE embedded systems letters.* 2013;5(3):34–37.
10. Automotive Fleet. 70 Percent of Drivers Using Smartphones. https://www.automotive-fleet.com/129558/70-of-drivers-use-smartphones-says-at-t-study.Accessed On: 18-Aug-2018.
11. de Graaff Ramon. Controlling your Connected Car. https://pure.tue.nl/ws/files/47037140/799539-1.pdf.Accessed On: 18-Aug-2018.
12. Jia Yunhan Jack, Zhao Ding, Chen Qi Alfred, Mao Z Morley. Towards Secure and Safe Appified Automated Vehicles. *IEEE Intelligent Vehicles Symposium (IV'17), Los Angeles, CA, USA, June 11-14.* 2017;:705–711.
13. Schweppe Hendrik, Roudier Yves. Security and privacy for in-vehicle networks. *2012 IEEE 1st International Workshop on Vehicular Communications, Sensing, and Computing (VCSC), Seoul, Korea(South), Jun 18-18.* 2012;:12–17.
14. Miller Charlie, Valasek Chris. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA.* 2015;:1–91.
15. Murvay Pal-Stefan, Groza Bogdan. Security shortcomings and countermeasures for the SAE J1939 commercial vehicle bus protocol. *IEEE Transactions on Vehicular Technology.* 2018;67(5):4325–4339.
16. Wolf Marko, Weimerskirch André, Paar Christof. Secure in-vehicle communication. *Embedded Security in Cars.* 2006;:95–109.
17. Mandal Amit Kr, Cortesi Agostino, Ferrara Pietro, Panarotto Federica, Spoto Fausto. Vulnerability analysis of Android auto infotainment apps. *15th ACM International Conference on Computing Frontiers, CF 2018, Ischia, Italy, May 08-10.* 2018;:183–190.
18. Panarotto Federica, Cortesi Agostino, Ferrara Pietro, Mandal Amit Kr, Spoto Fausto. Static Analysis of Android Apps Interaction with Automotive CAN. *The 3rd International Conference on Smart Computing and Communication, SmartCom 2018, Waseda University, Tokyo, Japan, Dec 10-12.* 2018;.
19. Cousot Patrick, Cousot Radhia. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, Jan 17-19.* 1977;:238–252.
20. Cortesi Agostino, Ferrara Pietro, Pistoia Marco, Tripp Omer. Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications. *VMCAI 2015, Mumbai, Jan. 12-14, 2015. LNCS vol. 8931.* 2015;:61–79.

21. QARK . Quick Android Review Kit - A tool for automated Android App Assessments. https://github.com/linkedin/qark.Accessed On: 18-Aug-2018.

22. Spoto Fausto. The Julia Static Analyzer for Java. *International Static Analysis Symposium, Edinburgh, UK, September 8-10.* 2016;:39–57.

23. dex2jar . dex2jar. https://github.com/pxb1988/dex2jar.Accessed On: 18-Aug-2018.

24. Apktool . Apktool. https://ibotpeaches.github.io/Apktool/.Accessed On: 18-Aug-2018.

25. Google . Android Auto App Quality Guidelines. https://developer.android.com/docs/quality-guidelines/auto-app-quality.Accessed On: 18-Aug-2018.

26. Ernst M. D., Lovato A., Macedonio D., Spiridon C., Spoto F.. Boolean Formulas for the Static Identification of Injection Attacks in Java. *Proc. of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20), Suva, Fiji, November 24-28.* 2015;9450:130–145.

27. Satam Pratik, Pacheco Jesus, Hariri Salim, Horani Mohommad. Autoinfotainment Security Development Framework (ASDF) for Smart Cars. *2017 International Conference on Cloud and Autonomic Computing (ICCAC), Tucson, AZ, USA, September 18-22.* 2017;:153–159.

28. McClure Stuart. Caution: malware ahead. https://trid.trb.org/view/1255020.Accessed On: 18-Aug-2018.

29. Mazloom Sahar, Rezaeirad Mohammad, Hunter Aaron, McCoy Damon. A Security Analysis of an In-Vehicle Infotainment and App Platform. *WOOT.* 2016;:1–12.

30. Jaiswal Gaurav. Android in-vehicle infotainment system (AIVI). *International Journal of Innovative Research in Electronics and Communications (IJIREC).* 2014;1(4):12–16.

31. Macario Gianpaolo, Torchiano Marco, Violante Massimo. An in-vehicle infotainment software architecture based on google android. *IEEE International Symposium on Industrial Embedded Systems, SIES'09, Lausanne, Switzerland, July 8-10.* 2009;:257-260.

32. Wiese Emily E, Lee John D. Auditory alerts for in-vehicle information systems: The effects of temporal conflict and sound parameters on driver attitudes and performance. *Ergonomics.* 2004;47(9):965–986.

33. Heikkinen Jani, Mäkinen Erno, Lylykangas Jani, Pakkanen Toni, Väänänen-Vainio-Mattila Kaisa, Raisamo Roope. Mobile devices as infotainment user interfaces in the car: contextual study and design implications. *15th international conference on Human-computer interaction with mobile devices and services, MobileHCI '13, Munich, Germany, August 27-30.* 2013;:137–146.

34. Udovicic Ksenija, Jovanovic Nenad, Bjelica Milan Z. In-vehicle infotainment system for android OS: User experience challenges and a proposal. *2015 IEEE 5th International Conference on Consumer Electronics-Berlin (ICCE-Berlin), Berlin, Germany, September 6-9.* 2015;:150–152.

35. Andersson Torbjörn, Warell Anders, Holmlid Stefan, Ölvander Johan. Desirability in the development of In-Car Infotainment Systems. *Interact 2011: 13th IFIP TC13 Conference on Human-Computer Interaction, Lisbon, Portugal, September 5-9.* 2011;:1–6.

36. Paupiah Pravin Selukoto. Vehicle security and forensics in Mauritius and abroad. *2015 International Conference on Computing, Communication and Security (ICCCS), Daegu University, Gyongsan, Korea. Nov 6-6.* 2015;:1–5.

37. Kun Andrew L, Boll Susanne, Schmidt Albrecht. Shifting gears: User interfaces in the age of autonomous driving. *IEEE Pervasive Computing.* 2016;15(1):32–38.

38. Kim Ho-Yeon, Choi Young-Hyun, Chung Tai-Myoung. Rees: Malicious software detection framework for meego-in vehicle infotainment. *2012 14th Int. Conference on Advanced Communication Technology (ICACT), Phoenix Park, PyeongChang, Korea(South), Feb 19-22.* 2012;:434–438.

39. Nisch Patrick. Security Issues in Modern Automotive Systems. *Citeseer.* 2011;:1–6.

40. Koscher K., Czeskis A., Roesner F., et al. Experimental Security Analysis of a Modern Automobile. *31st IEEE Symposium on Security and Privacy (S&P 2010), Berleley/Oakland, California, USA, May 16-19.* 2010;:447–462.

41. Avatefipour O., Hafeez A., Tayyab M., Malik H.. Linking Received Packet to the Transmitter through Physical-Fingerprinting of Controller Area Network. *IEEE Workshop on Information Forensics and Security (WIFS'17).* 2017;:1–6.

42. Wang Q., Sawhney S.. VeCure: A Practical Security Framework to Protect the CAN Bus of Vehicles. *4th International Conference on the Internet of Things (IOT'14), Cambridge, Massachusetts, USA, Oct 6-8.* 2014;:13–18.

43. Jaisingh Kushal, El-Khatib Khalil, Akalu Rajen. Paving the way for Intelligent Transport Systems (ITS): Privacy Implications of Vehicle Infotainment and Telematics Systems. *6th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications, Malta, November 13-17.* 2016;:25–31.

44. Giro Sergio. Android Developers Blog: Security "Crypto" provider deprecated in Android N https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html.Accessed On: 18-Aug-2018.

45. JuliaSoft . JuliaSoft. https://www.juliasoft.com/.Accessed On: 18-Aug-2018.

46. Bryant R.. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Survey.* 1992;24(3):293–318.

47. JDGUI . JDGUI. http://jd.benow.ca/.Accessed On: 18-Aug-2018.

**How to cite this article:** Mandal A. K., Panarotto F., Cortesi A., Ferrara P., and Spoto F. (2018), Static Analysis of Android Auto Infotainment and ODB-II Apps, *Softw Pract Exper*, *2018;00:0–0.*