Vulnerability Analysis of Android Auto Infotainment Apps

Amit Kr Mandal  
Università Ca’ Foscarì Venezia, Italy  
amitmandal.nitdgp@gmail.com

Agostino Cortesi  
Università Ca’ Foscarì Venezia, Italy  
cortesi@unive.it

Pietro Ferrara  
JuliaSoft Srl, Verona, Italy  
pietro.ferrara@juliasoft.com

Federica Panarotto  
University of Verona, Italy  
federica.panarotto@gmail.com

Fausto Spoto  
University of Verona, Italy  
fausto.spoto@univr.it

ABSTRACT

With over 2 billion active mobile users and a large array of features, Android is the most popular operating system for mobile devices. Android Auto allows such devices to connect with an in-car compatible infotainment system, and it became a popular choice as well. However, as the trend for connecting car dashboard to the Internet or other devices grows, so does the potential for security threats. In this paper, a set of potential security threats are identified, and a static analyzer for the Android Auto infotainment system is presented. All the infotainment apps available in Google Play Store have been checked against that list of possible exposure scenarios. Results show that almost 80% of the apps are potentially vulnerable, out of which 25% poses security threats related to execution of JavaScript.

CCS CONCEPTS

• Security and privacy → Abstract Interpretation; Static Analysis

KEYWORDS

Android Auto Security, Android Auto, Invehicle Infotainment System, Abstract Interpretation, Static Analysis

ACM Reference Format:

1 INTRODUCTION

Modern infotainment systems have evolved from a way to control the stereo or navigation system to be the hub of many vehicle functions such as telephone handling, data communication, vehicle setup and climate control. Moreover, with the increasing demand for more connected vehicles, and widespread use of smartphones, in-car entertainment is getting more and more sophisticated. Its ability to connect to a smartphone allows it to provide innovative capacities (e.g., messaging and audio apps). However, infotainment systems are not all created equal: every other car manufacturer and even electronic control unit (ECU) producer comes with its own infotainment system: Ford with SYNC and MyFord Touch [10], Toyota with Entune [31], Cadillac with the Cadillac User Experience (CUE) [4], Fiat Chrysler with Uconnect [32], and so on. To ease up technology landscape and reduce the vendor locking, software companies, such as Google and Apple, come up with infotainment systems. Given their popularity in the mobile phone market, many car manufacturers now support Apple CarPlay [3] and Android Auto [11]. Both are fairly similar and pump a small portion of the mobile phone’s experience into a car’s built-in infotainment system, allowing one to access some of the smartphone functions with a look and feel that is similar to that of the mobile phone. Therefore, the ability to run infotainment on a phone with almost 5 billion mobile users sets Android Auto as the most popular choice for the automotive infotainment.

Android Auto offers a slick and informative interface, inspired by Google Now, with the same card-based menu that is part of Google’s unified design language. Various Google approved apps are published in the marketplace, with the necessary driver-safety measures in place. Android Auto also allows one to interact with multiple devices connected to the car. This in turn leaves automobiles in a potentially vulnerable state in front of adversaries, as it provides many attack surfaces from multiple connections such as cellular, Wi-Fi, Bluetooth etc. If adversaries gain access to the infotainment system, they can play with the safety-critical functions: they can alter the vehicle’s electronic ID, jam with the radio-based systems, including the navigation system, spoof sensor data, interfere with control units, master data and firmware/software, just to name a few examples. Besides the risk of being attacked by adversaries from the outside world, Android Auto infotainment apps can be very harmful if they distract the attention of the driver. Thus, securing the Android Auto infotainment system has become essential, especially in today’s sociotechnical landscape, where 70% of drivers engage in infotainment activities while behind the wheel.

To address the security and privacy issues of the automotive infotainment system, various approaches have been proposed. McAfee analyzes emerging risks in automotive system security [23]. Pauriah et al. [26] and Bordoni et al. [20] discuss in detail the various security threats posed by the use of Android-based infotainment systems. Checkoway et al. [5] and Miller et al. [24] provide a comprehensive experimental approach to show that security of the modern automotive can be greatly compromised by interfering with Bluetooth, Wi-Fi and telematics signals. Some of the articles...
try to solve the security and privacy issues by using cryptographic
techniques, such as de Graaff et al. [8]. Others use a secure de-
velopment environment [17, 28]. However, most approaches in
the literature have only shown the issues, but never proposed a
feasible solution to the problem. In contrast to that, QARK [27]
provides a comprehensive static analysis tool for Android apps. It
looks for a wide range of standard mobile vulnerabilities, such as
WebViews, Broadcast, Cryptography etc. However, it does not pro-
vide solutions for Android Auto apps, as it does not cover Android
Auto specific vulnerabilities, such as GPS location, media auto-play,
image display, etc. as specified by Google [12].

Therefore, this article introduces a static analysis approach based
on abstract interpretation [6, 7] to discover software vulnerabili-
ties in Android Auto infotainment apps. There are already many
static analyzers able to analyze Java source code and find bugs
or inefficiencies. However, most of them are based on syntactical
analyses (Checkstyle, Coverity, FindBugs, PMD) or use theorem
proving with some simplifying hypotheses. Unfortunately, these
tools do not support technologies such as XML inflation, and this
affects the construction of the control flow graph of an Android app.
Instead, the Julia static analyzer [29] performs a semantic sound
analysis, and this is why we chose to develop our analyses on top
of it. Our analyses are based on the quality parameters specified by
Google for the infotainment system [12]. The overall architecture
of our system is the following. First, the apps are reverse engineered
through dex2jar [9] (to extract the Java bytecode) and apktool [2]
(to extract the Android manifest). The manifest is then used to
determine the entry points for parsing the Java bytecode of the
apps.

The analyzer is based on Android API 25. It works on the parsed
bytecode. If it detects a vulnerability, it issues a warning message
with its detail. These are programming patterns that our analyzer
identifies as potentially dangerous, since they are violations of
programming best practices that could lead to code that features
some exploitable software vulnerability. However, their actual ex-
plotiability is not the goal of our analyzer. This means that, as usual
in automatic static analysis, warnings should be checked, manu-
ally, by the programmer, to understand if they are actual security
problems or just false alarms.

To check the effectiveness of our system on Android Auto infor-
tainment software, available infotainment apps have been collected
from the Google Play store [30] (Snapshot of July, 2017). These are
then analyzed with the devised static analyzer based on abstract
interpretation. The results show that about 80% of the apps are po-
tentially vulnerable, out of which 25% poses security threats related
to execution of JavaScript.

The rest of the paper is organized in five sections. Section 2
discusses related literature. Section 3 provides a brief introduction
about the Julia static analyzer. Section 4 details the architecture of
the Android Auto analyzer. Section 5 summarizes and discusses the
experimental results. Section 6 concludes.

2 RELATED RESEARCH

The Android-based In Vehicle Infotainment (IVI) system has been
the focus of automobile research during the last decade. Several sci-
entific articles approach the development [15, 21], performance [34]
and user experience [13, 33] of the Android infotainment system.
On the other side, some researchers discussed the desirable features
of the Android-based car infotainment system [1]. However, only a
few papers face the issues related to the security and privacy of
Android-based auto infotainment system.

McAfee, in partnership with Wind River and ESCRYPT, released
a report called “Caution: Malware Ahead” [23] analyzing emerg-
ring risks in automotive system security and the security of ECUs
that have become omnipresent in modern automotive. The study
shows that an ECU connected to the infotainment system some-
how facilitates cybercriminal activities, such as remotely unlocking
and starting a car via cell phone, disabling a car remotely, tracking
a driver’s location, activities and routines, stealing personal data
from a Bluetooth system, disrupting navigation systems, disabling
emergency assistance, etc.

Jia et al. [17] introduced the concept of an app-based autonomous
vehicle (AV) platform providing the development framework to
third-party developers. However, to address safety and security
issues, they proposed an enhanced app-based AV design schema
called AVGUARD. This primarily focuses on mitigating the threats
posed by the use of untrusted code, by leveraging the theories of ve-
icle evaluation field and program analysis techniques. Finally, the
study sketches a guideline and suggests practices for the improve-
ment of future automotive apps. Paupiah et al. [26], and Bordonali
et al. [20] discussed in detail the various security threats posed by
the use of Android-based infotainment system. Whereas, Kim et
al. [19] analyzed an access control for IVI and proposed the Re-
stricted Execution Environment System (REES), to protect a mobile
handset connected to the car. REES is a malware detection system
able to analyze programs and provide mobility at the same time.
Besides this, Nisch [25] provided an insight into different aspects
of security vulnerabilities of the automotive ECU. In this work, a
detailed analysis of the threats related to the various ECU units,
such as tire pressure monitoring system (TPMS), global positioning
system (GPS), keyless entry system, on-board diagnostics (OBD-
II), audio system, Bluetooth connectivity and cell phone interface,
has been carried out. Results show that these units individually or
collectively induce serious threat to the security of the car.

Thus, researchers in industry and academia have demonstrated
the possibility of intruding safety critical components of an automo-
BILE. In this regard, Mazloom et al. [22] analyzed the vulnerabilities
of the apps, protocols and underlining IVI implementations through
the smartphone connected to the cars infotainment system. For this
purpose, they considered an IVI system that supports the Mirror-
Link protocol and comes with the 2015 model of a major automotive
manufacturer. They demonstrated that vulnerabilities in the Mir-
rorLink protocol used in infotainment systems could potentially
facilitate an attacker sending malicious messages on the vehicle’s
internal network from the connected smartphone.

However, in 2015, a serious security vulnerability in automobile
was demonstrated by Miller et al. [24], which obliged the manu-
facturer to recall 1.4 million vehicles. They were able to remotely
hack into the car and immobilize it while driving in a highway
traffic causing unintended acceleration and even disable the car’s
brakes by sending carefully crafted messages on the vehicle’s CAN
bus. In this process, they used the vehicle’s infotainment system

to access the ECU that sends legitimate commands to other ECU components.

Again, Schweppe et al. [28] presented an architecture capable of monitoring data flow into the automobile’s CAN network. This approach enhances vehicle security by using taint tracking tools along with a security framework capable of dynamically tagging data flows within or among the control units. They also implemented a prototype to prevent from damaging the on-board system using buffer overflow. Further, it shows the applicability of transport tags among network nodes by extending the communication payload. However, in this approach the overhead is quite high because of the tainted process. This makes it not suitable for real-time environments.

Beside security, very little attention was dedicated to protect the privacy of automotive customers. De Graaff et al. [8] discussed the enforcement of a higher level of privacy by using cryptographic techniques. They identified 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. Instead, Jaisingh et al. [14] provided an overview of how personal information flows through typical infotainment and telematics systems. They also identified potential privacy threats to drivers and provide security recommendations.

3 VULNERABILITIES IN ANDROID AUTO

Today’s Android ecosystem is a complex open network of collaborating companies. In addition to the Google’s operating system, more than 176 open source projects are widely used in the Android platform. Moreover, many hardware manufacturers and network providers customize Android to meet their requirements. This leaves the system in a vulnerable state. In particular, the Application Programming Interface (API) of Android Auto allows a developer 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, and the attacker repurposes those API elements to break the intended model. In particular, our analyses target the following vulnerabilities.

(1) **External File Access Detection**: Files created on 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. Further, the apps that use external storage should perform input validation when handling data from external storage, as it would contain executable files and data from any untrusted source, which can cause serious damage to the automobiles. The following code snippet shows the untrusted use of an external directory:

```java
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 protections enforce that only the app that created a file on the internal storage can access it. However, some apps do use modes MODE_WORLD_WRITEABLE or MODE_WORLD_READABLE for IPC files, thus bypassing this restriction. They also exploit the ability to load and control the data format. With world readable enabled, intruders can load malicious data and steal private information from the cars dashboard or from the smartphone, by using the app. This code snippet shows this malicious practice:

```java
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**: To provide additional protection for sensitive data, local files are often encrypted by using a key that is not directly accessible to the app. For example, keys can be placed in a keystore and protected with a password. However, this does not protect data in a root compromised system that can monitor the user inputting the password. The following code snippet depicts the usage of a keystore: if PasswordProtection(), load() and getPrivateKey() get tainted, then the security of the system is compromised:

```java
KeyStore.ProtectionParameter protParam = new KeyStore.PasswordProtection(password);
KeyStore.PrivateKeyEntry pkeEntry = (KeyStore.PrivateKeyEntry) ks.getEntry("privateKeyAlias", protParam);
PrivateKey myPrivateKey = pkeEntry.getPrivateKey();
javax.crypto.SecretKey mySecretKey = new KeyStore.SecretKeyEntry(myPrivateKey);
ks.setEntry("secretKeyAlias", skEntry, protParam);
try (FileOutputStream fos = new FileOutputStream("newKeyStoreName")) {
    ks.store(fos, password);
}
```

(4) **Content Providers**: Content providers are a structured storage mechanism that can be limited to a given app or exported to all apps. If one does not intend to provide other apps with access to the content provider, this should be reported in the manifest as android.exported=false. If instead that attribute is set to true, then other apps are allowed to access the data, which might leave the app in a vulnerable state. In addition, it is also important to check the usage and taintedness of the addPermission() function in the production code, as it takes a string that concisely expresses to a user the security decision that must be made. The following code snippet depicts the dynamic addition of that permission:

```java
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) **IP Networking**: For automotive apps, it is important to make sure that HTTPSURLConnection is used for sensitive sensor
data. Moreover, to handle sensitive IPC, some apps use local-host network ports. This is potentially dangerous, as these interfaces are also accessible to other apps in the infotainment system.

(6) Using WebView: WebView deals with HTML and JavaScript. Improper use of WebView instances can lead to major web security issues such as cross-site scripting or JavaScript injection. In this regard, usage of setJavaScriptEnabled(), and addJavaScriptInterface() can leave an app open to various attacks. In WebView, enabling JavaScript means that it is now susceptible to XSS. Thus if an app uses the following code snippet then one should inspect the rendered page for taintedness:

```java
public class JavaScriptAttack extends Activity {
    WebView wv = new WebView(getApplicationContext());
    wv.setWebChromeClient(new WebChromeClient() {
        public boolean onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {
            callback.invoke(origin, true, false);}
    });
    wv.getSettings().setJavaScriptEnabled(true);
    wv.addJavaScriptInterface(new jsInvokeclass(), "attack");
    wv.loadUrl("http://www.malware.com/atk.html");}
```

Again, addJavaScriptInterface() allows JavaScript to invoke operations that are usually particular to the Android apps. If used, then it should expose addJavaScriptInterface() only to web pages from which all input is trustworthy. Otherwise, untrusted JavaScript can invoke Android methods within your app. According to Google’s recommendation, calls to addJavaScriptInterface() should only expose to JavaScript which is contained within the app. The following code snippet depicts the security issues with addJavaScriptInterface():

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

(7) GPS Location Detector: WebChromeClient.onGeolocationPermissionsShowPrompt() method is called by the WebView to obtain permission to disclose the user’s location to JavaScript. If it is implemented, the app should seek permission from the user. However, the following code snippet will always grant the permission. Thus, location service of the automobile is greatly compromised:

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

(8) Background Download: While downloading file, if the storage location is not explicitly defined, the programmer uses the DownloadManager.openDownloadedFile() method with the ID value stored in preferences, to get a ParcelFileDescriptor, which can be turned into a stream the app can read from. Further, without a specific destination, downloaded files are in the shared download cache. In this case, the system retains the right to delete them at any time to reclaim space. This in-turn leaves the app in a vulnerable state. Thus, it is necessary to check the taintedness of the following functions for any potential security breach:

- 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.

(9) Media Autoplay: According to the security standards set by Google Android Auto media app, one should never autoplay a media. However, the following code snippet will always play media files:

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

(10) CarMode: Google appstore lists Android Auto apps in a separate category. Apps belonging to this category should explicitly use the UiModeManager.enableCarMode("true") function to declare it as an Android auto app. This prevents users from installing apps that are not suitable for automobiles.

(11) Voice Commands: Android Auto apps should allow users to control audio content playback with voice actions. This will provide a hands-free experience to the user, while driving and listening to audio content in Android Auto. To enable voice-enabled playback controls, Android Auto apps must enable the hardware controls by setting these flags in the app’s MediaSession object:

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

(12) 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 sort of web API or online service. Thus, apps should not use such services. The following code snippet shows an example:

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

(13) Displaying an HTML Page: The simplest case is displaying an HTML page or image by supplying the URL of the resource to the WebView. This can be a source of injection in
Android/Android Auto apps, especially when JavaScript is enabled. The following code snippet shows one such incident:

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

Further, a malicious program can take advantage of the WebViewClient .shouldOverrideUrlLoading() callback to intercept and monitor log user activity. This is a serious privacy breach. The following code snippet shows this scenario:

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

(14) Media Advertisement: Android Auto apps should not push notifications. To prevent Android Auto from displaying a notification, the media metadata key android.media.metadata.ADVERTISEMENT must be set to 1. The following code snippet keeps the provision of displaying the notification open, which is against the Google standards for Android Auto:

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

We present here the complete list of checks since we wanted to assess broadly the various types of possible vulnerabilities even if they are not (actually) present in existing applications as shown by our experimental results in Section 3.

4 A STATIC ANALYZER FOR ANDROID AUTO

In this work, a static analyzer for Android Auto apps is developed by extending the Julia static analyzer [18]. The devised checker relies on the heap and call graph abstractions performed by Julia, but it implements completely novel property checks targeting the possible vulnerable points discussed in Section 3.

4.1 The Julia Static Analyzer

The Julia static analyzer [18] uses abstract interpretation for the analysis and verification of Java bytecode [29]. 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 as well. 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 field or methods that are accessed/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 threats to security.

4.2 Architecture of the System

To ensure the security of modern automobiles that use the Android Auto infotainment system, we developed a static analyzer for Android Auto. It uses abstract interpretation for analysis and verification of Java bytecode. The analyzer uses the Julia bytecode representation and is developed as a checker inside the Julia framework. Figure 1 depicts a schematic diagram of the Android Auto static analyzer. Here, firstly the apps are reverse engineered with dex2jar [9] and apktool [2]. These tools extract the app manifest and jar files from the apk file. The manifest is then used to determine the entry points for analyzing the Java bytecode. The Android Auto checker based on the vulnerabilities described in the previous section is then implemented for Android API 25 and applied to the parsed bytecode to detect presence of those vulnerabilities in the Android Auto apps. Thus, selection of entry points and construction of Android Auto checkers play the most important role in this process. The following subsections describe these in detail.

4.2.1 Entry Points. In static analysis, the entry points are a crucial element for the soundness and coverage of the analysis. 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 reflection. Therefore, the Android Auto static analyzer should start the analysis from all such handlers. Moreover, every Android app uses an AndroidManifest.xml file, that describes the important
properties such as program structure, permissions, user interface parameters etc. Moreover, to get complete information about the event handlers, the analyzer must consider how they receive input at runtime, by looking at XML files, such as layout files, that are inflated at runtime.

Besides the generic event handlers in activities, services, broadcast receivers, content providers, WebView services, FileStorage, DownloadManager etc. special attention is given to the MediaBrowser and Messaging services as Android Auto currently supports audio playback and messaging for the music app. Thus, locating the classes responsible for these two activities are of prime importance.

Android Auto browses audio track listings by using the MediaBrowser service. The audio apps must declare this service in their manifest. This allows the dashboard system to discover this service and connect to the app. Figure 2 depicts the extraction process of the MediaBrowser service class from the Android Auto apk. Here, it first searches for the class responsible for enabling the Android.media.browse.MediaBrowserService service from the manifest. Then, this class is used as an entrypoint to parse the bytecode. Moreover, classes responsible for creating a MediaSession service are also considered as an entrypoint.

Similarly, for the messaging services all the 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 entrypoint. The similar process is followed for locating other types of activities and services.

4.2.2 Analysis Process. The entrypoints are then used to build a semantic model of the bytecode execution. The Android Auto checkers work on this parsed bytecode to detect bugs. The working principle of the checker involves searching for the identified vulnerable API implementation in the production code. If such implementation is encountered, then it is necessary to check the taintedness of the implementation. For this purpose, the JVM stack for that API call is accessed, where all the producers of values passed to that API call are traced. If the arguments to these contributors are constants, then the implementation is secure; otherwise, it is at risk of possible attacks. This process is repeated for all the identified vulnerabilities. Figure 3 depicts the process.

5 EXPERIMENTAL RESULTS

The Android Auto apps have been analyzed through the analyzer described in the previous section. For this purpose, the Android Auto apps have been manually collected from the Google Play Store. These apps have been then converted into Java bytecode with dex2jar. Further, the manifest file has been extracted from the apk. These jar files, along with their manifest, have been used as input to the analyzer. Experiments have been performed on Windows OS with an Intel Xeon machine running at 2.66 GHz and 8 gigabytes of RAM. Table 1 shows the service specific entry points for the individual apps. Here, the numbers in column a refer to the number of classes where event handling such as click, focus etc. are implemented. Similarly, the numbers in columns b, c, d, e, f, g, h represent the number of classes where the functionalities related to services, broadcast receiver, content providers, media browser, WebView services, file storage and text messaging are implemented. These cover almost all the user interactive implementations in the apps.

The classes selected as entry points are then used to build the static call graph of the app. Table 2 depicts the reachability within an app’s code in terms of Lines Of Code (LOC), along with entrypoints. Here, total LOC have been calculated by decompiling the app with a Java decompiler [16]. The results show that, with the selected entry points, we reached 50% to 75% of the code. This moderate percentage in reachability is because, while parsing the bytecode, we mainly considered the classes that are accessible from the entrypoints. Further, we also ignored local variables. Keeping these facts in mind, the selected entry points cover the vast majority of the code in an app, where the logic is implemented. This prohibits us from checking every class file present in an app, which in turn reduces the analysis time.

The results related to the vulnerabilities are shown in Table 3 and plot in Figure 4. The numbers there indicate how many threats have been detected for a specific type of vulnerability. From the table, it is visible the analyzer detected vulnerabilities related to
the JavaScript execution and file and cache directory access for almost 80% of the apps; out of these, 25% possess JavaScript execution threat. However, apps like smartaudiobookplayer, icq mobile, itunestoppodcastplayer, jetaudio and spotify do not show any single threat to the infotainment system. Whereas, the auto analyzer generates warnings for the rest of the apps. Further, out of 14 identified security threats, the majority of the apps only show threats related to the JavaScript execution and access to external storage devices. This is because the majority of the Android Auto apps use Google Now services, in one way or another, for location, event processing, voice input, notification services etc. thus making the apps interaction with the user more secure.

To check the severity of the warnings, we looked into the de-compiled source code. For this purpose, the podcastaddict app was chosen, as it contains 11 warning and 212 entrypoints. For demonstrating such an example, the com.bambnetworks.

```java
package com.bamnetworks.mobile.android.gameday.activities;

import com.bamnetworks.
```

mobile.android.gameday.activities.BlackoutActivity class is considered, that is accessible because of the activity type entrypoint of the analyzer. Here, both the setJavaScriptEnabled() and the addJavaScriptInterface() functions are called. Interestingly, from the following code snippet it can be seen that the first argument of the addJavaScriptInterface() function is not constant. It is initialized multiple times, even once with an intent call. Further, the setJavaScriptEnabled() function is set to true, which allows execution of JavaScript:

```java
package com.bambnetworks.mobile.android.gameday.activities;

public class BlackoutActivity extends AtBatDrawerActivity{
    public void onCreate(Bundle paramBundle){
        ... 
        paramBundle = getIntent().getStringExtra("zip");
        ...
        paramBundle = new
        BlackoutActivity.BlackoutMapJavaScriptInterface(this, dl, d2);
        this.BlackoutMap.addJavaScriptInterface(paramBundle,
        "android");
        this.BlackoutMap.getSettings().setJavaScriptEnabled(true);
    ... }
```
Table 3: Result of the Android Auto checker in Julia.

<table>
<thead>
<tr>
<th>Sl No</th>
<th>App Name</th>
<th>Javascript</th>
<th>External File</th>
<th>External Cache</th>
<th>Total</th>
<th>Analysis Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>smartaudiobookplayer</td>
<td>3.3.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>79.17</td>
</tr>
<tr>
<td>2</td>
<td>abcrewrs</td>
<td>5.12.67</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>891.72</td>
</tr>
<tr>
<td>3</td>
<td>itunerfire</td>
<td>4.2.10</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>721.92</td>
</tr>
<tr>
<td>4</td>
<td>audible</td>
<td>2.12.0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>614.09</td>
</tr>
<tr>
<td>5</td>
<td>audiobooks</td>
<td>6.4.4</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>161.49</td>
</tr>
<tr>
<td>6</td>
<td>pockastidetid</td>
<td>3.43.8</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>111.83</td>
</tr>
<tr>
<td>7</td>
<td>MLB.com At Bat</td>
<td>5.0.6</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>601.08</td>
</tr>
<tr>
<td>8</td>
<td>testplus</td>
<td>7.0.7</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>927.59</td>
</tr>
<tr>
<td>9</td>
<td>icq mobile</td>
<td>6.13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1691.43</td>
</tr>
<tr>
<td>10</td>
<td>tinwortopodcastplayer</td>
<td>2.8.10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>375.72</td>
</tr>
<tr>
<td>11</td>
<td>jetaudio</td>
<td>8.2.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>199.93</td>
</tr>
<tr>
<td>12</td>
<td>override</td>
<td>3.6.2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>442.28</td>
</tr>
<tr>
<td>13</td>
<td>sonyericsson</td>
<td>8.5.2.7</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>159.91</td>
</tr>
<tr>
<td>14</td>
<td>spotify</td>
<td>8.4.11.1283</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3086.08</td>
</tr>
<tr>
<td>15</td>
<td>stitcher radio</td>
<td>5.9.5</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>455.51</td>
</tr>
<tr>
<td>16</td>
<td>deezer</td>
<td>5.8.4.66</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1004.13</td>
</tr>
<tr>
<td>17</td>
<td>fm player</td>
<td>3.7.4.0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>373.12</td>
</tr>
<tr>
<td>18</td>
<td>kik (11.29.0.061)</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>717.12</td>
</tr>
<tr>
<td>19</td>
<td>beyondpod</td>
<td>4.2.16</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>283.39</td>
</tr>
<tr>
<td>20</td>
<td>npr</td>
<td>1.7.2.2</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>451.66</td>
</tr>
<tr>
<td>21</td>
<td>tunein player</td>
<td>18.3.1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>523.18</td>
</tr>
<tr>
<td>22</td>
<td>sevendigital</td>
<td>6.3.1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>412.18</td>
</tr>
<tr>
<td>23</td>
<td>JDGUI</td>
<td>1.4 (2014)</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
<tr>
<td>24</td>
<td>Uconnect</td>
<td>2013</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
<tr>
<td>25</td>
<td>Connectivity</td>
<td>2015</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
<tr>
<td>26</td>
<td>JDGUI</td>
<td>1.4 (2014)</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
<tr>
<td>27</td>
<td>Uconnect</td>
<td>2013</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
<tr>
<td>28</td>
<td>Connectivity</td>
<td>2015</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>721.92</td>
</tr>
</tbody>
</table>

ACKNOWLEDGMENTS

Work partially supported by Regione Veneto within the FSE project “Static analysis for the safety and security of Android systems for automotive infotainment”, and by CINI Cybersecurity National Laboratory within the project Filiera Sicura.

REFERENCES