Vulnerability Analysis of Android Auto Infotainment Apps

Amit Kr Mandal†, Agostino Cortesi†, Pietro Ferrara‡, Federica Panarotto§, Fausto Spoto§

†Università Ca' Foscari Venezia, Italy; ‡JuliaSoft Srl, Verona, Italy; §University of Verona, Italy
amitmandal.nitdgp@gmail.com;cortesi@unive.it;pietro.ferrara@juliasoft.com
federica.panarotto@gmail.com;fausto.spoto@univr.it

ABSTRACT

With almost 5 billion 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 a 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 vulnerable, out of which 25% poses security threats related to execution of JavaScript.

KEYWORDS

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

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]. Paupiah *et al.* [26] and Bordonali *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 development 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 provide 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 vulnerabilities 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 exploitability is not the goal of our analyzer. This means that, as usual in automatic static analysis, warnings should be checked, manually, 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 infotainment 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 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 scientific 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 emerging 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 somehow 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 vehicle evaluation field and program analysis techniques. Finally, the study sketches a guideline and suggests practices for the improvement 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 Restricted 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 automobile. 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 MirrorLink protocol and comes with the 2015 model of a major automotive manufacturer. They demonstrated that vulnerabilities in the MirrorLink 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 manufacturer 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:

```
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:

```
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 compromized system that can monitor the user inputing 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 compromized:

```
KeyStore.ProtectionParameter protParam = new
     KeyStore.PasswordProtection(password);
KeyStore.PrivateKeyEntry pkEntry =
     (KeyStore.PrivateKeyEntry)
ks.getEntry("privateKeyAlias", protParam);
PrivateKey myPrivateKey = pkEntry.getPrivateKey();
javax.crypto.SecretKey mySecretKey;
KeyStore.SecretKeyEntry skEntry = new
     KeyStore.SecretKeyEntry(mySecretKey);
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:

```
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 *localhost* 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:

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

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 apk. The following code snippet depicts the security issues with addJavaScriptInterface().

```
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");}}
```

(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 compromized:

```
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:

```
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);}}
```

(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:

```
mSession.setFlags(MediaSession.FLAG_HANDLES_MEDIA_BU
TTONS|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:

```
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:

```
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);}}
```

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

```
public class MyActivity extends Activity {
 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){
    Uri request = Uri.parse(url);
    return true; }}}
```

(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:

```
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());}
```

We present here the complete list of checks since we wanted to asses 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 5.

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.

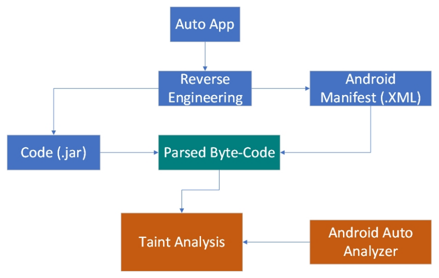


Figure 1: System architecture of the Android Auto static analyzer.

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 on 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.

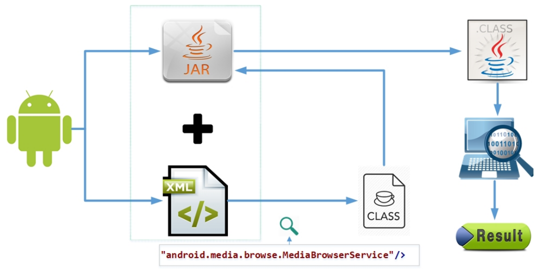


Figure 2: Entrypoint for the analyzer: audio playback classes.

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.

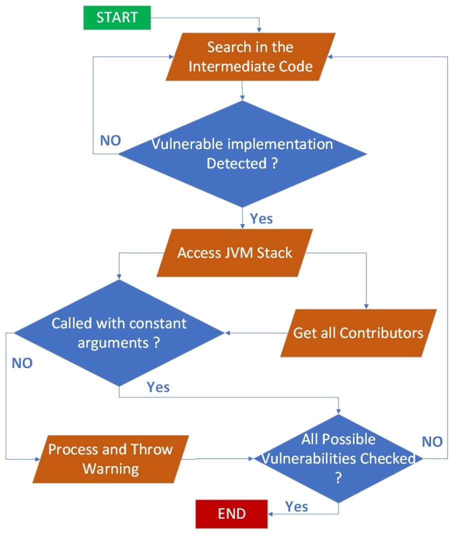
Android Auto browses audio track listings by using the MediaBrowser service. The audio apps must declare this service in their manifest. This allows the dashboard system to discover this service and connect to the app. Figure 2 depicts the extraction process of the MediaBrowser service class from the 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 a 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



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

Figure 3: Working principle of the Android Auto analyzer.

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 entry points. 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 decompiled 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.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:

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

In this app, the analyzer also comes up with an external file access warning. From the following code snippet, it can be seen that the app can save a critical file in the external device, which can be deleted by the system, user or other app to regain space. This could leave the app in a vulnerable state:

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

Table 1: Entrypoints for the analyzed apps.

Sl No	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 (3.12.07)	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	sonyericsson (8.5.A.2.7)	12	20	12	10	0	0	0	1	55
14	spotify (8.4.11.1283)	240	112	36	8	0	8	25	5	434
15	stitcher radio (3.9.8)	88	26	24	2	0	9	0	0	149
16	simpleradio (2.2.5.1)	24	14	14	4	1	6	0	0	63
17	deezer (5.4.8.46)	162	36	30	12	0	23	21	2	286
18	fm player (3.7.4.0)	70	38	24	6	0	1	0	5	144
19	kik (11.29.0.17461)	40	280	10	8	0	8	12	0	358
20	beyondpod (4.2.16)	62	12	18	4	0	6	0	2	104
21	npr (1.7.2.2)	44	42	14	6	2	1	0	1	110
22	tunein player (18.3.1)	94	32	44	20	0	13	0	3	206
23	sevendigital (6.69.226)	66	86	22	2	0	27	0	3	206

a) Activities, b) Services c) Broadcast Receivers, d) Content Providers, e) Media Browser, f) WebView Service, g) File Storage, h) Text Messaging

Table 2: Reachability analysis.

Sl no	App Name	Entrypoint	Reachable LOC	Total LOC	Reachability Percentage
1	smartaudiobookplayer (3.3.5)	54	76426	109902	69.54
2	abcnews (3.12.07)	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	sonyericsson (8.5.A.2.7)	55	115262	178542	64.56
14	spotify (8.4.11.1283)	434	370505	773512	47.90
15	stitcher radio (3.9.8)	149	239605	389262	61.55
16	simpleradio (2.2.5.1)	63	189583	341488	55.52
17	deezer (5.4.8.46)	286	400704	791891	50.60
18	fm player (3.7.4.0)	144	173005	311657	55.51
19	kik (11.29.0.17461)	358	359798	492574	73.04
20	beyondpod (4.2.16)	104	130085	271281	47.95
21	npr (1.7.2.2)	110	214492	361362	59.36
22	tunein player (18.3.1)	206	281758	438734	64.22
23	sevendigital (6.69.226)	206	214219	375949	56.98

6 CONCLUSION AND FUTURE WORK

As far as we know, this is the first static analysis for Android Auto apps based on a formal basis such as abstract interpretation that has been systematically applied to the apps published in the Google Play store. The experimental results show that five apps do not pose a single threat to the infotainment system, while 80% of the apps exposes some warnings, out of which 25% apps are open to JavaScript attacks, leaving the infotainment system at serious risk.

Our future work includes the extension of the analyzer towards detecting the vulnerabilities of safety critical systems such as the CAN bus, the climate control, the telematics etc. Further, we are also

Table 3: Result of the Android Auto checker in Julia.

Sl No	App Name	JavaScript	External File	External Cache	Total	Analysis Time
1	smartaudiobookplayer (3.3.5)	0	0	0	0	79.17
2	abcnews (3.12.07)	3	5	5	13	891.72
3	itunerfree (4.2.10)	6	5	5	16	721.92
4	audible (2.12.0)	0	2	2	4	614.09
5	audiobooks (4.64)	0	3	3	6	161.49
6	podcastaddict (3.43.8)	4	5	2	11	711.83
7	MLB.com At Bat (5.6.0)	5	2	2	9	601.08
8	textplus (7.0.7)	0	2	6	8	927.59
9	icq mobile (6.13)	0	0	0	0	1693.43
10	itunestoppodcastplayer (2.8.10)	0	0	0	0	375.72
11	jetaudio (8.2.3)	0	0	0	0	199.93
12	overdrive (3.6.2)	0	2	1	3	442.28
13	sonyericsson (8.5.A.2.7)	0	6	4	10	159.91
14	spotify (8.4.11.1283)	0	0	0	0	3086.08
15	stitcher radio (3.9.8)	0	5	3	8	416.37
16	simpleradio (2.2.5.1)	0	3	3	6	435.51
17	deezer (5.4.8.46)	2	4	3	9	1004.13
18	fm player (3.7.4.0)	0	5	1	6	373.12
19	kik (11.29.0.17461)	0	3	3	6	717.12
20	beyondpod (4.2.16)	0	4	4	8	283.39
21	npr (1.7.2.2)	0	6	4	10	451.66
22	tunein player (18.3.1)	4	3	3	10	523.18
23	sevendigital (6.69.226)	0	3	2	5	412.18

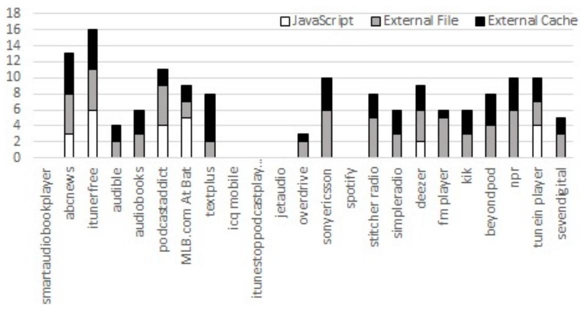


Figure 4: Number of warnings in the Apps.

contacting car manufacturers and companies developing Android Auto apps, for applying our analysis to real world apps.

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 FilieraSicura.

REFERENCES

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

[2] Apktool. Accessed On: 18-Jan-2018. Apktool. https://ibotpeaches.github.io/Apktool/. (Accessed On: 18-Jan-2018).

[3] Apple. Accessed On: 18-Jan-2018. Apple CarPlay The ultimate copilot. https://www.apple.com/ios/carplay/. (Accessed On: 18-Jan-2018).

[4] Cadillac. Accessed On: 18-Jan-2018. Cadillac User Experience | Infotainment. http://www.cadillac.com/cadillac-user-experience.html. (Accessed On: 18-Jan-2018).

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces.. In *USENIX Security Symposium.* San Francisco.

[6] Agostino Cortesi, Pietro Ferrara, Marco Pistoia, and Omer Tripp. 2015. Datacentric Semantics for Verification of Privacy Policy Compliance by Mobile Applications. In *VMCAI 2015, Mumbai, Jan. 12-14, 2015. LNCS vol. 8931.* 61–79.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977.* 238–252.

[8] Ramon de Graaff. 2015. Controlling your Connected Car. (2015).

[9] dex2jar. Accessed On: 18-Jan-2018. dex2jar. https://github.com/pxb1988/dex2jar. (Accessed On: 18-Jan-2018).

[10] Ford. Accessed On: 18-Jan-2018. SYNC 3 - Smart hardware, Smart software, Smart design. https://www.ford.com/technology/sync/. (Accessed On: 18-Jan-2018).

[11] Google. Accessed On: 18-Jan-2018. Android AUto. https://www.android.com/auto/. (Accessed On: 18-Jan-2018).

[12] Google. Accessed On: 18-Jan-2018. Google. https://developer.android.com/develop/quality-guidelines/auto-app-quality.html. (Accessed On: 18-Jan-2018).

[13] Jani Heikkinen, Erno Mäkinen, Jani Lylykangas, Toni Pakkanen, Kaisa Väänänen-Vainio-Mattila, and Roope Raisamo. 2013. Mobile devices as infotainment user interfaces in the car: contextual study and design implications. In *Proceedings of the 15th international conference on Human-computer interaction with mobile devices and services.* ACM, 137–146.

[14] Kushal Jaisingh, Khalil El-Khatib, and Rajen Akalu. 2016. Paving the way for Intelligent Transport Systems (ITS): Privacy Implications of Vehicle Infotainment and Telematics Systems. In *Proceedings of the 6th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications.* ACM, 25–31.

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

[16] JDGUI. Accessed On: 18-Jan-2018. JDGUI. http://jd.benow.ca/. (Accessed On: 18-Jan-2018).

[17] Yunhan Jack Jia, Ding Zhao, Qi Alfred Chen, and Z Morley Mao. 2017. Towards Secure and Safe Appified Automated Vehicles. *arXiv preprint arXiv:1702.06827* (2017).

[18] JuliaSoft. Accessed On: 18-Jan-2018. JuliaSoft. https://www.juliasoft.com/. (Accessed On: 18-Jan-2018).

[19] Ho-Yeon Kim, Young-Hyun Choi, and Tai-Myoung Chung. 2012. Rees: Malicious software detection framework for meego-in vehicle infotainment. In *Advanced Communication Technology (ICACT), 2012 14th Int. Conference on.* IEEE, 434–438.

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

[21] Gianpaolo Macario, Marco Torchiano, and Massimo Violante. 2009. An in-vehicle infotainment software architecture based on google android. In *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on.* IEEE, 257–260.

[22] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. 2016. A Security Analysis of an In-Vehicle Infotainment and App Platform.. In *WOOT.*

[23] Stuart McClure. 2013. Caution: malware ahead. *Vision Zero International* (2013).

[24] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA* 2015 (2015).

[25] Patrick Nisch. 2011. Security Issues in Modern Automotive Systems. (2011).

[26] Pravin Selukoto Paupiah. 2015. Vehicle security and forensics in Mauritius and abroad. In *Computing, Communication and Security (ICCCS), 2015 International Conference on.* IEEE, 1–5.

[27] QARK. Accessed On: 18-Jan-2018. Quick Android Review Kit - A tool for automated Android App Assessments. https://github.com/linkedin/qark. (Accessed On: 18-Jan-2018).

[28] Hendrik Schweppe and Yves Roudier. 2012. Security and privacy for in-vehicle networks. In *Vehicular Communications, Sensing, and Computing (VCSC), 2012 IEEE 1st International Workshop on.* IEEE, 12–17.

[29] Fausto Spoto. 2016. The Julia Static Analyzer for Java. In *International Static Analysis Symposium.* Springer, 39–57.

[30] Google Play Store. Accessed On: 18-Jan-2018. Apps for Android Auto. https://play.google.com/store/apps/collection/promotion_3001303_android_auto_all?hl=en. (Accessed On: 18-Jan-2018).

[31] Toyota. Accessed On: 18-Jan-2018. Entune 3.0. https://www.toyota.com/owners/resources/entune. (Accessed On: 18-Jan-2018).

[32] Uconnect. Accessed On: 18-Jan-2018. Uconnect Systems for Chrysler, FIAT, Jeep, Dodge, and Ram Trucks. https://www.driveuconnect.com/. (Accessed On: 18-Jan-2018).

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

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