CHALMERS
UNIVERSITY OF TECHNOLOGY

On the road with third-party apps: Security analysis of an in-vehicle app platform

Downloaded from: https://research.chalmers.se, 2021-08-31 11:51 UTC

Citation for the original published paper (version of record):

Eriksson, B., Groth, J., Sabelfeld, A. (2019)
On the road with third-party apps: Security analysis of an in-vehicle app platform
VEHITS 2019 - Proceedings of the 5th International Conference on Vehicle Technology and Intelligent

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology.
It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004.
research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

On the Road with Third-party Apps: Security Analysis of an In-vehicle App Platform

Benjamin Eriksson, Jonas Groth and Andrei Sabelfeld
Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

Keywords: In-vehicle App Security, API Security, Program Analysis for Security, Infotainment, Information Flow Control, Android Automotive.

Abstract: Digitalization has revolutionized the automotive industry. Modern cars are equipped with powerful Internet-connected infotainment systems, comparable to tablets and smartphones. Recently, several car manufacturers have announced the upcoming possibility to install third-party apps onto these infotainment systems. The prospect of running third-party code on a device that is integrated into a safety critical in-vehicle system raises serious concerns for safety, security, and user privacy. This paper investigates these concerns of in-vehicle apps. We focus on apps for the Android Automotive operating system which several car manufacturers have opted to use. While the architecture inherits much from regular Android, we scrutinize the adequateness of its security mechanisms with respect to the in-vehicle setting, particularly affecting road safety and user privacy. We investigate the attack surface and vulnerabilities for third-party in-vehicle apps. We analyze and suggest enhancements to such traditional Android mechanisms as app permissions and API control. Further, we investigate operating system support and how static and dynamic analysis can aid automatic vetting of in-vehicle apps. We develop AutoTame, a tool for vehicle-specific code analysis. We report on a case study of the countermeasures with a Spotify app using emulators and physical test beds from Volvo Cars.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

The modern infotainment system, often consisting of a unit with a touchscreen, is mainly used for helping the driver navigate, listening to music or making phone calls. In addition to this, many users wish to use their favorite smartphone apps in their cars. Thus, several car manufacturers, including Volvo, Renault, Nissan and Mitsubishi (Volvo Car Group, 2018; Renault–Nissan Alliance, 2018), have chosen to use a special version of Android for use in cars, called *Android Automotive* (Google Inc., 2018b). Other manufacturers such as Volkswagen (Volkswagen, 2018) and Mercedes-Benz (Mercedes-Benz, 2018) are instead developing their new in-house infotainment systems. In contrast to the in-house alternatives, Android Automotive is an open platform with available information and code. This justifies our focus on Android Automotive apps.

Android Automotive. For the manufacturers, a substantial benefit of using an operating system based on Android is gained from relying on third-party developers to provide in-vehicle apps. A multitude of popular apps already exists on the Android market, which can be naturally converted into Android Automotive apps. Further, Android Automotive is a stand-alone platform that does not require a connected smartphone, contrary to its competitors MirrorLink (MirrorLink, 2009), Apple CarPlay (Apple, 2014) and Android Auto (Google Inc., 2014).

Safety, Security, and Privacy Challenges. While third-party apps boost innovation, they raise serious concerns for safety, security, and user privacy. Indeed, it is of paramount importance that the platform both safely handles these apps while driving and also safeguards the user's privacy-sensitive information against leakage to third parties. Figure 1 gives a flavor of real-life safety concerns, by showing a user comment on a radio app with almost half a million downloads. The user points out that they had to stop driving when a shockingly loud ad was suddenly played, adding that ads "shouldn't attempt to kill you" (Warren, 2018).

While the Android Automotive architecture inherits much from regular Android, a key question is whether its security mechanisms are adequate for in-

Eriksson, B., Groth, J. and Sabelfeld, A.
On the Road with Third-party Apps: Security Analysis of an In-vehicle App Platform.
DOI: 10.5220/0007678200640075
In *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS 2019)*, pages 64-75
ISBN: 978-989-758-374-2
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

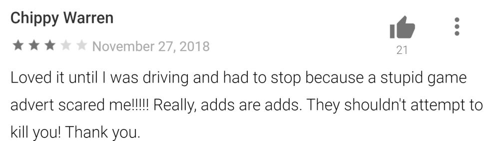

Figure 1: Top comment on a radio app. A user was shocked by the volume of an ad and had to stop driving.

vehicle apps. However, compared to the setting of a smartphone, in-vehicle apps have obvious safety-critical constraints, such as neither being able to tamper with the control system nor being able to distract the driver. Further, car sensors provide sources of private information, such as location and speed or sound from the in-vehicle microphone. In fact, voice controls are encouraged for apps in infotainment systems, as to help keep the driver's hands on the wheel, opening up for audio snooping on users by malicious apps. A recent experiment done by GM collected location data and radio listening habits from its users with the goal of creating targeted radio ads (Detroit Free Press, 2018). This clearly highlights the value of user data in vehicles. Similar data could potentially be collected by apps using the radio API to record the current station (Gampe, 2018). Thus, a key question is whether Android's security mechanisms are adequate for in-vehicle apps.

Android Permissions Android's core security mechanism is based on a *permission model* (Google Inc., 2018d). This model forces apps to request permissions before using the system resources. Sensitive resources such as camera and GPS require the user to explicitly grant them before the app can use them. In contrast, more benign resources such as using the Internet or NFC can be granted during installation. However, there are several limitations of this model with implications for the in-vehicle setting. From a user's perspective, these permissions are often hard to understand. Porter Felt et al. (Porter Felt et al., 2012b) show that less than a fifth of users pay attention to the permissions when installing an app, and even a smaller fraction understands the implications of granting them. Understanding the implications of giving permissions is even harder. There are immediate privacy risks, such as an app having permission to access the car's position and the Internet can potentially leak location to any third party. More advanced attacks would only need access to the vehicle speed. This may not seem like a privacy issue but by knowing the starting position, likely the user's home address, and speed, it is possible to derive the path that the car drives (Gao et al., 2014).

Analyzing Android Automotive Security. To the best of our knowledge, this is the first paper to analyze application-level security on the Android Automotive infotainment system. To assess the security of the Android Automotive app platform, we need to extend the scope beyond the traditional permissions.

Attack Surface. For a systematic threat analysis, we need to analyze the attack surface available to third-party apps. This includes analyzing what effects malicious apps may have on the functions of the car, such as climate control or cruise control, and on the driver. We demonstrate *SoundBlast*, representative of disturbance attacks, where a malicious app can shock the driver by excessive sound volume, for example, upon reaching high speed. We also demonstrate availability attacks like *Fork bomb* and *Intent Storm.* which render the infotainment system unusable until it is rebooted. Further, we explore attacks related to the privacy of sensitive information, such as vehicle location and speed, as well as in-vehicle voice sound. We show how to exfiltrate location and voice sound information to third parties. In order to validate the feasibility of the attacks, we demonstrate the attacks in a simulation environment obtained from Volvo Cars. Based on the attacks, we derive exploitable vulnerabilities and use the Common Vulnerability Scoring System (CVSS) (FIRST.Org Inc., 2018) to assess their impact.

To address these vulnerabilities, we suggest countermeasures of permissions, API control, system support, and program analysis.

Permissions. We identify several improvements to the permission model. This includes both introducing missing permissions, such as those, pertaining to the location and the sound system in the car, as well as making some permissions more fine-grained. For location, there are ways to bypass the location permission by deriving the location from IP addresses. At the same time, the location permission currently allows all or nothing: either sharing highly accurate position information or not. The former motivates adding missing permissions, while the latter motivates making permissions more fine-grained. We argue that for many apps, like Spotify or weather apps, low-precision in the location, e.g. city-level, suffices.

API Control. In contrast to permissions, API control can use more information when decided to grant an app access to a resource. For example, using high-precision data could be allowed only once an hour, or during an activity like running. Our findings reveal that apps currently need access to the microphone in order to use voice controls. We deem this as breaking

the principle of least privilege (Saltzer and Schroeder, 1975). To address this, we argue for full mediation, so that apps subscribe to voice commands mediated by the operating system, rather than having access to the microphone. Similarly, location data can also be mediated to limit the precision and frequency of location requests, making it possible to adhere to the principle of least privilege. These scenarios exemplify countermeasures we suggest to improve API controls for in-vehicle apps.

System. We argue for improvements to the operating system in order to protect against apps using too much of the system's resources. Malicious apps can cause the system to become unresponsive or halt, either by recursively creating new processes or coercing other system processes to use up all the resources. The countermeasures consist of limiting the number of requests an app can make, limiting the resources system processes can use, or completely blocking some capabilities for third-party apps, like creating new processes.

Code Analysis. While previous methods protect the device from malicious apps, our vision is to also be able to stop the apps before they make it to the device. This can be accomplished by analyzing the code in the app store, before the app is published. This does not only protect against malicious apps but also poorly written apps that fail to adhere to security best practices. This could, for example, include apps not using encryption for data transmissions, which is currently a big problem (Razaghpanah et al., 2017). Other problems include vulnerable apps with high privileges being exploited by malicious apps or colluding malicious apps sharing data over covert channels. Thus, we investigate how static and dynamic program analysis can be leveraged to address the vulnerabilities.

We design and develop AutoTame, our own static analysis tool for detecting dangerous use of APIs, including the new automotive APIs. Further, we explore several state-of-the-art techniques, based on tools like FlowDroid (Arzt et al., 2014) and We are Family (Balliu et al., 2017).

Threat Model. The threat model in this paper defines the attacker as being able to install one or more apps, with the victim's permission, on their infotainment system. Similar to previous research (Schlegel et al., 2011), we assume that the victim is more inclined to install an app that asks for fewer permissions. This means that, while one app with access to both Internet and GPS might be considered suspicious, two apps, one with access to the Internet, the other with access to GPS, would be more acceptable. Such a model incentivizes apps to collude and share information over covert channels. Using this model we analyze how much damage can be done by a user mistakenly installing malicious apps.

Case Study. An ideal evaluation of our countermeasures would be a large-scale of apps from an app store, in the style of the studies on Google Play, e.g. (Porter Felt et al., 2011a; Arzt et al., 2014; Enck et al., 2014). Unfortunately, Android Automotive is at this stage an emerging technology with no apps yet publicly available for a study of this kind. Nevertheless, we have been granted access to Infotainment Head Unit emulators and physical test beds from Volvo Cars allowing us to perform a case study with an in-vehicle app version of Spotify. We use this infrastructure to evaluate our countermeasures.

Impact. At the same time, an early study of Android Automotive security has its advantages. Because our analysis comes at an early phase of Android Automotive adoption by car manufacturers, it has higher chances for impact. We have reported our findings to both Volvo Cars that participated in our experiments and Google. We are in contact with both on closing the vulnerabilities we point out and on experimenting with the countermeasures.

Contributions. The paper offers the following contributions:

- We present an attack surface for third-party in-vehicle apps, identifying classes of disturbance, availability, and privacy attacks (Section 3).
- We propose countermeasures, based on fine-grained permissions, API control, system support, and information flow (Section 4).
- We overview prominent representatives of techniques and tools for detecting security and privacy violations in third-party apps (Section 4.4).
- We present our own static analysis tool, AutoTame[1], for detection of dangerous API usage (Section 4.4).
- We evaluate the countermeasures on a case study with the in-vehicle app Spotify (Section 5).

2 BACKGROUND

As cars become more connected and their infotainment systems more powerful, people expect the car

[1]Our implementations are available online on https://www.cse.chalmers.se/research/group/security/autotame/

to interact in a seamless way with their other devices. In contrast to most other personal devices, a software bug in a car can have lethal consequences. For example, in 2015 Miller and Valasek (Miller and Valasek, 2015) showed that it was possible to remotely take over a 2014 Jeep Cherokee by exploiting their infotainment system Uconnect. More recently, in May 2018, researchers found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars which made it possible to gain control of the CAN buses in the vehicle (Tencent Keen Security Lab, 2018). These type of attacks show that remote take over attacks of connected vehicles is a possibility and a real threat.

Attackers do not necessarily need to take control over the braking or steering system to endanger or distract the driver. For example, an attacker can make a malicious infotainment app that disturbs or shocks the driver at a certain speed level. In order to shock the driver, the app may, for example, play loud music or rapidly flash the screen.

In addition to security, privacy is also a concern as cars become more capable of collecting data about their users. In accordance with the new EU regulation, GDPR (European Commission, 2016), the user has to be informed about how the data is used and agree to their data being used in the described way. Previous research projects have explored the possibility to automatically track and analyze how privacy-sensitive information is leaked from Android apps (Reyes et al., 2017), either deliberately through advertisement networks or inadvertently through insecure communication means (Razaghpanah et al., 2017).

2.1 Experimental Setup

With access to Volvo Cars internal testing equipment, both the attacks and countermeasures were tested on their infrastructure. In particular, the code is tested on Volvo's *Infotainment Head Unit emulators (IHU)* emulators and physical test beds. All of the Android code is developed for Android SDK version 26 and 27, which corresponds to Android 8.0 and 8.1.

2.2 Automatic Analysis of Android Apps

Automatically analyzing Android apps can be done through two major strategies, static analysis or dynamic analysis. Static analysis only considers the code while in dynamic analysis the code is executed and the program's behavior is analyzed. Whichever method is chosen, a decision on what to look for in the analysis has to be made. In this paper, two tracks are evaluated, how privacy-sensitive information flows through the app and scanning apps for common vulnerabilities.

2.3 Android Automotive

Today, the Android system is officially used in all types of devices, from phones and tablets to watches, TVs and soon cars (Google Inc., 2018a). Android Automotive is a version of Android developed specifically for use in cars. It is essentially Android with a User Interface (UI) adapted for cars and a number of car specific APIs. The car specific APIs allow for control over vehicle functions, such as the heating, ventilation, and air conditioning (HVAC), and reading of sensor data, e.g. speed, temperature and engine RPM (Google Inc., 2018b). Android Automotive is not to be confused with Android Auto which is already available on the market today. Unlike Android Auto, Automotive is a completely stand-alone system that is not dependent on a smartphone. In Android Auto, apps run on the users Android phone which then renders content on a screen in the car. The apps and the Android system thus runs separated from the car.

2.4 Android's Permission Model

The Android operating system controls access to many parts of the system, such as camera, position and text messages, through permissions. These permissions can be of one of four types; *normal*, *dangerous*, *signature* or *signatureOrSystem*. The first two are the most common and can be granted to any third-party app. Normal permissions give isolated accesses with minimal risk for the system and user, these are automatically granted by the operating system. Dangerous permissions, on the other hand, give accesses to private user data and control over the device that may harm the user. These permissions have to be explicitly granted by the user on a per-application basis. Both Android's *coarse* and *fine* location permissions are examples of dangerous permissions, since both supply high precision data. The difference between them is that *fine* location has access to the GPS while *coarse* uses cell towers and WiFi access points. Finally, there are the *signature* and *signatureOrSystem* permissions, which requires the app to be pre-installed or cryptographically signed (Google Inc., 2018c).

2.5 Covert Channels

A *covert channel*, as defined by Lampson, is a communication channel between two entities that are not intended for information transfer (Lampson, 1973). In Android, a number of different covert channels exist that use both hardware attributes and software functions to communicate. Apps can, for example, communicate by reading and setting the volume, sending special intents or cause high and low system load (Marforio et al., 2012; Schlegel et al., 2011).

3 ATTACKS

This section focuses on the implementation decisions regarding the attacks presented in Table 1. The category and asset columns in the table give an understanding of what the attack is targeting. More specifically, the asset is what the attack is trying to take control over. In the case of denial-of-service (DoS) attacks, this is usually some type of resource. Privacy attacks, on the other hand, try to acquire and exfiltrate data such as speed or location. User interaction and permission are used to judge how easy the attack is to execute. The values are finally combined to create a severity score based on the Common Vulnerability Scoring System (CVSS3) (FIRST.Org Inc., 2018). A shortcoming of CVSS3 is that possible physical damage or safety risks are not considered in the scoring. Distraction vulnerabilities, like the one exploited by SoundBlast, and other automotive vulnerabilities will be underrated. These shortcomings are currently being revised for CVSS3.1 (Dugal, 2018). The exact vectors and scores for each attack are presented in Table 3. Table 2 present the same attacks together with suitable countermeasures to mitigate the underlying vulnerabilities.

3.1 Disturbance

SoundBlast. The SoundBlast attack relies heavily on the `AudioManager` class in Android. This class supplies functions which are used to control the volume of different audio streams in Android. Cars also have the more specific `CarAudioManager`, however, this class requires special permissions. Different audio streams are used to differentiate between volumes, e.g. music volume, ringer volume, alarm volume, etc. A malicious app can use the permissionless audio API to max the volume and shock the driver. The attack is further improved by using a `ContentObserver` to listen for changes in volume and force the volume to the maximum as soon as it changes. Using the vehicle's sensors, the attacker can also design the attack to only active when traveling at high speeds.

Testing the SoundBlast attack shows that it is possible to set any volume on all the different audio streams in Android, without needing any permissions. In addition, the attack can also detect changes in volume and max the volume accordingly. The changes are also detected even if the driver uses the hardware controls on the IHU or steering wheel. Killing the app is the only way to regain control of the volume.

3.2 Availability

Fork Bomb. A fork bomb is a program that creates new instances of itself until the system runs out of resources, either freezing the device or force a reboot. While this might be acceptable on a phone, in a vehicle setting this is problematic. Since the IHU usually handles navigation, freezing the device might distract drivers trying to fix it, or frustrate them by having to stop and reboot.

Forking in Android is not possible by default, resulting in the need for a vulnerability to leverage in order to accomplish forking. Unlike previously successful fork bomb attacks on Android (Armando et al., 2012), our attack takes an application-level approach by creating a shell, which in turn has the power to fork itself. By using `exec` to run `sh -s`, a new shell is created, which can execute the fork bomb.

When testing this attack it is able to fully grind both the emulator and test bed to a halt, requiring a power cycle to regain control. It is thus able to render the infotainment system unusable until the system is rebooted.

Intent Storm. The intent storm attack uses Android intents to continuously restart the app itself. Similar to the fork bomb presented in section 3.2, the intent storm attack tries to use up all the CPU resources, making the IHU unusable. The difference, however, is that the intent storm does not use the resources itself, but rather forces another system process, the *system_server*, to use up all resources. The fast activity switching required is made possible with threads and intents. As soon as the app starts, it spins up 8 threads which all ask Android to start its own main activity. Using multiple threads increases the pressure on the *system_server*, making the device less responsive.

During the tests, the *system_server* process was forced by the attack to use 100% of the CPU, making the IHU unusable. Similar to the fork bomb in section 3.2, this would grind the IHU to a halt. However, in some cases, the IHU would automatically restart after a few minutes.

Table 1: The attacks are divided into three different categories. Which asset and permissions the attacks affects and requires are listed along with the needed user interaction.

Name	Category	Asset	User interaction	Permission	Severity
SoundBlast	Disturbance	Driver's attention	Start app	None	Medium[a]
Fork bomb	DoS	CPU resources	Start app	None	Medium
Intent storm	DoS	CPU resources	Start app	None	Medium
Permissionless speed	Privacy	Current speed	Start app	None	Low
Permissionless exfiltration	Privacy	Data Exfiltration	Start app	None	Low
Covert channel	Privacy	Data Exfiltration	Start app	Channel dependent	Low

[a] The score is subject to the limitation of CVSS3 on lacking support for physical damage and safety risks (Dugal, 2018).

3.3 Privacy

Permissionless Speed. In Android Automotive, apps have direct access to the current speed. However, since speed is privacy sensitive it requires a permission. By combining other permissionless sensor values, such as the current RPM and gear, and knowledge about the wheel size, the speed can be derived. The effectiveness of this attack does depend on the sampling frequency of the sensors. The hardware test beds only contained the IHU and not the full car, meaning that the efficiency of the attack is yet to be tested.

Permissionless Exfiltraion. The Android permission model clearly states that any app wanting communicate on a network requires Internet permission. However, by using intents it is possible to force another app with Internet permission to leak the data. Different apps handle different intents, for example, the web browser will open URLs, music player opens music files, etc. While the implementation details differ depending on the app, the common procedure is to encode the data, split it into chunks and send a separate intent for each chunk.

Crafting an intent with data type `audio/wav` and using the URL http://evil.com/music.wav?d=[data], forces the music player to load the URL. The native Android music player is quite stealthy as it only shows a small popup with a play button will appear. By returning a malformed wav file from the server, the music player will only show a subtle error message.

If a web browser is used, the attacker can have the server redirect the request to a deep link, giving control back to the exfiltration app. Not only does this give the app the ability to leak more data, but it also enables two-way communication with the attacker's server, all without using the Internet permission.

In order to test this, a proof-of-concept code was developed that would record audio for five seconds and then upload it using the described method. The code only needs permission to record audio, but not to use the Internet. Testing this attack shows that it is possible to send data to the Internet without using the Internet permission. The attack was successful using Chrome, the standard music player, video player and image viewer. If the device has not been configured with a default application for opening the type of data, it will ask the user to pick one.

Covert Channels. Previous work on covert channels in Android have used both vibration and volume settings to transmit data between colluding apps (Schlegel et al., 2011). In addition, Android Automotive introduces some new APIs. In particular the new climate control API for temperature. Since the temperature is represented by a floating point value, the bandwidth is more than tenfold that of the volume settings. However, changing the temperature does currently require a signature permission, making it hard for third-party apps to acquire.

In contrast to previous work on covert channels, which relied on time synchronization, our attack is based on asynchronous messages. This forces the receiver to send an acknowledgment for each of the received values. While this lowers the bit rate, in contrast to synchronous communication, it greatly increases the reliability of the communication.

With this implementation, two apps can collude to leak privacy- sensitive information to the Internet. One app requests permission to privacy-sensitive information but not the Internet and then acts as a sender. The second app requests Internet permission but not permission to access any sensitive data. The second app can now receive sensitive information which it does not have permission for and leak it to the Internet.

4 COUNTERMEASURES

The vulnerabilities are very different in nature and, as such, the mitigation techniques differ. Some vulnerabilities can be mitigated by several different techniques while others can only be mitigated by one. An overview of the attacks together with mitigations for

the underlying vulnerabilities are presented in Table 2.

4.1 Permission

The current permission model can be improved both by adding new permissions for unprotected resources, and also by refining some very broad permission. The SoundBlast attack, from Section 3.1, relies on changing the volume through an API called *AudioManager* which does not require any permission. At the same time, there exists an API called *CarAudioManager*, which does require a permission. Cars usually have more advanced sound systems than phones so a different API with more settings does make sense as does the need for a permission. When conducting experiments with the emulator the *AudioManager* is usable by third-party apps, thus allowing an attacker to change the volume without any permission.

In addition to audio, Android allows apps to get the location of the device by using GPS. This can, for example, be used by apps to give weather information. However, due to these systems having high precision and allowing for real-time updates, apps often excessive information.

There are multiple methods for preserving the user's privacy while still maintaining an acceptable level of functionality in apps using location (Micinski et al., 2013; Fawaz and Shin, 2014). Which method is optimal is highly dependent on the type of information the app needs. A simple approach is to truncate location, effectively creating a grid of possible locations. A grid will better protect the privacy of the user, but at the same time degrade the functionality of some apps (Micinski et al., 2013). In order to handle apps like fitness trackers, which requires fast updates and high precision, truncation is not feasible. Fawaz and Shin (Fawaz and Shin, 2014) argue that in order to preserve privacy, a choice has to be made between tracking distance and speed, or tracking the path of the exercise. They present a method for tracking the distance and speed by supplying the exercise tracker with a synthetic route, that has correct distance and speed but a forged path. Furthermore, they argue that navigation apps with Internet access, usually used for real-time traffic information, are the hardest to handle since they can potentially leak the location. This problem could be solved by using state-of- the-art information flow tracking to ensure that the location is never leaked.

4.2 API Control

In some scenarios, permissions are not enough. This is usually the case when access to a resource can be abused over time. For example, in the current Android model, apps are allowed to record audio from the microphone at all times, as long as it has been granted the permission once. This means that a restaurant app that uses voice commands to find close by restaurants, can listen to everything the user says, at all times. Since voice commands are more prevalent in vehicles, where the user's focus is on driving, it is reasonable to believe that more in-vehicle apps will use this functionality. One solution to this problem is to use a voice mediator, which is a special service that has access to the microphone and allows for third-party apps to subscribe to certain keywords. The app would only receive sentences that contain the keywords it subscribed to, effectively removing its capabilities to eavesdrop. Similar to the voice mediation, the same method can be used for location. By using a location mediator apps can subscribe to arbitrary precision for location data. The mediator can also introduce a trade-off between the refresh rate and precision of the requests, mitigating real-time tracking.

4.3 System

Some problems are best solved at the operating system level. These problems include resource management, e.g. how much CPU time or memory an app should be allowed to use. Android already limits resource usage by apps to a great extent when it comes to memory and CPU However, some system processes, the *system_server* process in particular, can use all of the CPU, effectively starving the rest of the system. This lack of rate limiting was exploited in the intent storm attack in Section 3.2. While not tested, we speculate that this vulnerability could either be countered by rate limiting the CPU usage of the *system_server* process or limit incoming intents to the *system_server*.

Similar to CPU limiting, memory usage requires limitations too. When Android is running low on memory it will start to terminate apps in the background. This can result in the termination of apps that the user wants to run in the background, e.g. navigation apps. A possible method for ensuring that the navigation works while driving is to prohibit Android from terminating important apps. This protects against both malicious apps using up the memory, and legitimate memory hungry apps.

Akin to permissions, SELinux policies are policies which limit what the processes in an OS can

do. These policies play a crucial role in protecting the vehicle's subsystems from Android. The policies are suitable for specifying what an app is allowed to do. However, not how many times it can do it. As Bratus et al. (Bratus et al., 2011) explains, "SELinux does not provide an easy way to control the use of the fork operation once forking has been allowed in the program's profile", which shows that SELinux is not suited to stop attacks like fork bombing. While it might be infeasible in many situations, blocking forking altogether could be a solution.

4.4 Code Analysis

Automatic analysis techniques can be used to scan apps, both before installation and during runtime, to find vulnerabilities and block attacks. The following sections describe this in more detail.

Vulnerability Detection. Both AndroBugs (Lin, 2018) and QARK (LinkedIn Corporation, 2018) are tools that can be used to scan Android apps for known vulnerabilities. QARK is capable of finding many common security vulnerabilities in Android apps (Ibrar et al., 2017). QARK can, for example, find incorrect usage of cryptographic functions, trace intents and detect insecure broadcasts. In addition, QARK can also generate exploits for some of these vulnerabilities. While not able to generate exploits, AndroBugs can detect vulnerabilities based on heuristics in the code. For example, multiple dex files suggests a master key vulnerability (CVE-2013-4787) (MITRE, 2013). The tools work well together since AndroBugs can quickly scan multiple apps with heuristics and then QARK can perform a deeper analysis of the interesting apps.

AutoTame. To scan for dangerous use of the new automotive APIs, we developed a special tool built on the Soot framework, which can analyze both Java and Android bytecode. The tool has a list of dangerous APIs, e.g controlling the HVAC system, change audio volume or spawning shells. Using Soot, our tool decompiles the APK and analyses each function in the app while testing if it matches any of the ones in the list. AutoTame performs a full application analysis. The main advantage of this is that it does not require any entry point analysis. Compared to many other languages, Android apps do not have a single main function from which execution starts. Therefore a full analysis ensures that any dangerous use of an API is detected. However, without knowing the entry points, dead code could be flagged, potentially leading to false positives. In addition to only detecting if the volume is changed, AutoTame can also give extra warnings if the volume is set to a high numeric value or if `getStreamMaxVolume` is used. If a match is found the app can be removed or marked as potentially dangerous. The tool was able to flag the SoundBlast attack, as well as the fork bomb.

Taint Tracking. Taint tracking can help detect privacy leaks where sensitive information, such as the user's location, is being sent to a remote server. FlowDroid (Arzt et al., 2014) is a tool for static taint analysis on Android, that can detect these flows. The taint analysis works by tainting private sources of information, such as the user's location. If the location is written to a variable, then this variable also becomes tainted. If at a later time this tainted variable is written to a public sink, e.g an Internet connection, a leak from a private source to a public sink will be detected.

What makes FlowDroid special is its highly accurate modeling of Android's life cycles. This is important as an app can be started in many different ways. FlowDroid is also able to track leaks via button clicks and other UI events. Important for the car API used in this paper is that FlowDroid can track dynamically registered callback functions, which is used to establish the connection to the car.

In order to make FlowDroid fully functional with Android Automotive apps, we extended the tool with new sources and sinks. Some of the sources added were used to acquire the car's manufacturer, model and year. For sinks, we added functions for writing to the climate control APIs.

Observable Flows. Taint tracking is not always enough to find all privacy leaks. For this reason, a more powerful tool that can detect observable implicit flows is introduced. The *We are Family* paper by Balliu et al. (Balliu et al., 2017) presents a two-fold hybrid analysis solution. The first stage is a static analysis that transforms the application and adds monitors. These monitors will aid the dynamic analysis tool in the second stage to find implicit flows. The added monitors are in this case used to track the program counter label and analyze the current taint value, making it possible to detect potential leaks during runtime on the device. The dynamic tool developed in the paper is an extension of TaintDroid (Enck et al., 2014). By using the transformed program together with TaintDroid, observable implicit flows can be detected, something TaintDroid was not able to do.

Table 2: List of all developed attacks and which countermeasure(s) can be used to mitigate each attack the underlying vulnerabilities.

Attacks / Countermeasures	Permissions	Location granularity	SELinux	AutoTame	FlowDroid	We are Family	Rate limit
SoundBlast	✓			✓			
Fork bomb			✓	✓			
Intent Storm							✓
Permissionless speed		✓			✓	✓	
Permissionless exfiltration		✓			✓	✓	
Covert channels	✓	✓			✓	✓	

5 SPOTIFY CASE STUDY

To test some of the countermeasures, an in-depth case study was performed on the Spotify app. The motivation behind using Spotify is that it was the only third-party app available on the emulator and test bed, making it the most realistic app to test. It was also much larger in size than the proof-of-concept attacks. The larger size will show how well the methods handle real apps.

5.1 Permissions

The first analysis that has to be performed is to gather an understanding of the permissions the app uses. Spotify needs permission to Internet, Bluetooth and NFC, for data transfer. Furthermore, it also requires permission to change audio settings, run at startup, and prevent the device from sleeping. Since Spotify is a music streaming app that should be able to run in the background, as well as talk to other Bluetooth devices, these permissions seem innocuous. Shifting focus to the *dangerous* permissions, Spotify does require permission to read the accounts on the device, contacts stored on the device, the device ID, and information about current calls. It is not clearly motivated why this information is necessary, and while some connection between the Spotify user and the device user is reasonable, having access to all contacts seems excessive. Spotify does not ask for the location permission, instead, they use IP-addresses for location (Spotify, 2018). In addition, Spotify can also record audio and take pictures, as well as read and write access to the external storage. Taking pictures is necessary to scan QR-codes and the microphone will be used in Spotify's driving mode (Singleton, 2018). Access to external storage is reasonable since it allows for offline storage of music, however, it does include access to other photos and media files beyond Spotify's.

5.2 Vulnerability Detection

To ensure that the app does not have any known vulnerabilities QARK is used to scan the app. While QARK didn't find any severe vulnerabilities, it did find cases where a vulnerability could arise, e.g. by using a WebView in an older version of Android (API ≤ 18). Moreover, it also points out interesting entry-points into the app, one of them leading to a version of Spotify meant for another automotive system. In addition, a malicious third-party app can also send intents to Spotify to search and play arbitrary music, skip songs, or even crash the app. QARK did not find any vulnerabilities relating to the vehicle APIs, motivating the need for further analysis.

5.3 AutoTame

Using AutoTame, multiple warnings about both changing the volume and querying for max volume was found. Further manual analysis proved that the maximum volume was used directly to set the volume, as shown in Figure 2.

```
int i = this.c.getStreamMaxVolume(0);
this.c.setStreamVolume(0, i, 0);
```

Figure 2: Decompiled code setting volume to max.

5.4 Information Flow Analysis

The permissions give an upper bound on what the app is capable of doing. A more precise understanding of the app is achieved by analyzing it with FlowDroid, using implicit flow tracking. Using these settings the information flow analysis found 13 leaks in the app. One interesting leak was `getLastKnownLocation` being leaked into a dynamic receiver registration. As shown in Figure 3, FlowDroid was able to track the sensitive location through different assignments, function calls and control flows. While this case might be quite benign, as it only leaks one bit, it still shows the capabilities of the technique.

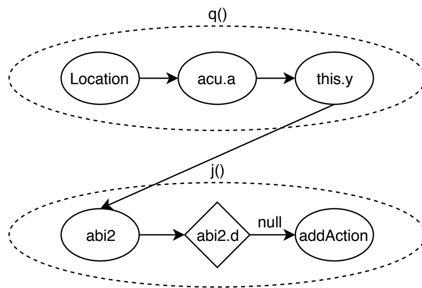

Figure 3: The publicly observable addAction function is implicitly dependent on the private location information.

The analysis also over-approximates some leaks, especially when the information being sent is based on information being received. A concrete example of this is when threads try to communication using `sendMessage` and `obtainMessage`. Since the obtained information could contain sensitive information, it is flagged as a leak. This could potentially be solved using dynamic information flow tracking.

5.5 Summary

To summarize these findings, we see that a more robust and at the same time more fine-grained permission model would be beneficial, as it would allow apps like Spotify to use lower precision location data instead of privacy-invading high precision data. In addition, vulnerability detection methods succeed in finding a bug that could be exploited to terminate Spotify. Finally, static analysis proved successful for automatically detecting privacy leaks.

6 RELATED WORK

Previous security and privacy research on vehicles have to a large extent focused on low-level problems relating to the internal components. Koscher et al. (Koscher et al., 2010) showed that with physical access to the CAN bus it is possible to control both the speedometer, horn and in-vehicle displays to distract the driver. Miller and Valasek (Miller and Valasek, 2015) gained similar access to the CAN bus, this time remotely. A similar vulnerability found in an infotainment system used in cars from Volkswagen was also recently discovered by researchers in the Netherlands (Computest, 2018). They showed that it was possible to connect to the car via WiFi to exploit a service running in the infotainment system to gain remote code execution system. The most recent study on attacks against vehicles was done by researchers at Tencent Keen Security Lab (Tencent Keen Security Lab, 2018), where they found multiple vulnerabilities in the infotainment system and Telematics Control Unit of BMW cars, resulting in control of the CAN buses.

A contribution of our paper is to show that even without access to the internal buses or exploiting low-level vulnerabilities, it is possible to cause distractions and leak private information.

A more high-level study was done by Mazloom et al. (Mazloom et al., 2016) where they conducted a security analysis of the MirrorLink protocol. MirrorLink allows smartphones to run apps on the cars infotainment system. Their analysis showed weaknesses in the MirrorLink protocol which could, amongst other things, allow malicious smartphone apps to play unwanted music or interfere with navigation. Mandal et al. (Mandal et al., 2018) showed that the similar system Android Auto have multiple problems that can be abused by third-party apps. For example, auto-playing audio when launching an app or showing visual advertisements, both of which are against Android Auto's quality policy. In our paper, we show similar attacks are possible on Android Automotive, however, without the requirement of the user's smartphone, since the malicious app runs on the infotainment system.

Intents, which is the main component in our exfiltration attack, are problematic for many reasons. Khadiranaikar et al. (Khadiranaikar et al., 2017) highlighted some of these problems, including how malicious apps can both steal information and compromise other apps using intents. Our paper builds on these ideas to develop new exfiltration methods for the Android Automotive platform.

There is a large body of work on Android permissions (Porter Felt et al., 2011b; Porter Felt et al., 2012a; Frank et al., 2012). As a representative example, a study on Android permissions by Porter Felt et al. (Porter Felt et al., 2011a) shows that many apps are using more permissions that they need, i.e. not adhering to the principle of least privilege. Other researches (Bugiel et al., 2013), also argue for the need of a more fine-grained model which can grant access to specific functions instead of full APIs or services. Extensions such as Apex (Nauman et al., 2010) have also been developed in order to supply end users with a more fine-grained model, capable of granting permissions based on user-specified policies. While our focus is on the specifics of the in-vehicle setting, we argue that many apps get access to excessive data due to the coarse granularity of the permission model itself. For example, a weather app or Spotify app only needs low-precision location, such as city level.

7 CONCLUSIONS

To the best of our knowledge, we have presented the first study to analyze application-level security on the Android Automotive infotainment system. Unfortunately, our analysis shows that in-vehicle Android apps are currently as secure as regular phone apps. We argue it is insufficient because in-vehicle apps can affect road safety and to some extent user privacy.

Our study of the attack surface available to third-party apps include driver disturbance, availability, and privacy attacks, for which there is currently no protection mechanisms in Android Automotive.

Consequently, it is important for car manufacturers that third-party apps are limited in their abilities to cause a considerable distraction for the driver. Additionally, there are a number of vehicle specific APIs, such as access to current gear and engine RPM, that is a cause for concern when it comes to user privacy.

To address the vulnerabilities that lead to these attacks, we have suggested the countermeasures of robust and fine-grained permissions, API control, system support, and program analysis.

We have designed and developed AutoTame, a tool for detecting dangerous vehicle-specific API usage. We have demonstrated that in-vehicle code analysis can be performed using AndroBugs and QARK, to detect known vulnerabilities, AutoTame to detect vehicle specific vulnerabilities and FlowDroid, with the additional vehicle specific sources and sinks, to detect privacy leaking apps.

We have evaluated the countermeasures with a Spotify app using an infrastructure of Volvo Cars.

ACKNOWLEDGEMENTS

Thanks are due to Henrik Broberg for orchestrating collaboration with Volvo Cars and for his valuable comments on earlier drafts of the paper. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) under the WebSec project and the Swedish Research Council (VR) under the PrinSec project.

REFERENCES

Apple (2014). Apple carplay. http://www.apple.com/ios/carplay/.

Armando, A., Merlo, A., Migliardi, M., and Verderame, L. (2012). Would you mind forking this process? a denial of service attack on android (and some countermeasures). In *IFIP*.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269.

Balliu, M., Schoepe, D., and Sabelfeld, A. (2017). We Are Family: Relating Information-Flow Trackers. In *European Symposium on Research in Computer Security*.

Bratus, S., Locasto, M. E., Otto, B., Shapiro, R., Smith, S. W., and Weaver, G. (2011). Beyond selinux: the case for behavior-based policy and trust languages.

Bugiel, S., Heuser, S., and Sadeghi, A.-R. (2013). Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*.

Computest (2018). Research paper: The connected car - ways to get unauthorized access and potential implications. Technical report.

Detroit Free Press (2018). Gm tracked radio listening habits for 3 months: Here's why. https://eu.freep.com/story/money/cars/general-motors/2018/10/01/gm-radio-listening-habits-advertising/1424294002/.

Dugal, D. (2018). List of potential improvements for cvss 3.1.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2014). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29.

European Commission (2016). Regulation (eu) 2016/679. http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf.

Fawaz, K. and Shin, K. G. (2014). Location privacy protection for smartphone users. In *SIGSAC '14*.

FIRST.Org Inc. (2018). Common vulnerability scoring system v3.0: User guide.

Frank, M., Dong, B., Porter Felt, and Song, D. (2012). Mining permission request patterns from android and facebook applications. In *ICDM '12*. IEEE.

Gampe, A. (2018). Radiotestfragment. https://android.googlesource.com/platform/packages/services/Car/+/4d1e3469cb2f285e7a4a864bd48a4c5177e7c83f/tests/EmbeddedKitchenSinkApp/src/com/google/android/car/kitchensink/radio/RadioTestFragment.java.

Gao, X., Firner, B., Sugrim, S., Kaiser-Pendergrast, V., Yang, Y., and Lindqvist, J. (2014). Elastic pathing: Your speed is enough to track you. In *ubicomp 2014*.

Google Inc. (2014). Android auto. https://www.android.com/auto/.

Google Inc. (2018a). Android. https://www.android.com/.

Google Inc. (2018b). Automotive. https://source.android.com/devices/automotive/.

Google Inc. (2018c). permission. https://developer.android.com/guide/topics/manifest/permission-element.html.

Google Inc. (2018d). Permissions overview. https://developer.android.com/guide/topics/permissions/overview.

Ibrar, F., Saleem, H., Castle, S., and Malik, M. Z. (2017). A study of static analysis tools to detect vulnerabili-

ties of branchless banking applications in developing countries. In *ICTD '17*.

Khadiranaikar, B., Zavarsky, P., and Malik, Y. (2017). Improving android application security for intent based attacks. In *IEMCON 2017*.

Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. (2010). Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*.

Lampson, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615.

Lin, Y.-C. (2018). Androbugs framework. https://github.com/AndroBugs/AndroBugs_Framework.

LinkedIn Corporation (2018). Qark. https://github.com/linkedin/qark.

Mandal, A. K., Cortesi, A., Ferrara, P., Panarotto, F., and Spoto, F. (2018). Vulnerability analysis of android auto infotainment apps. In *CF '18*. ACM.

Marforio, C., Ritzdorf, H., Francillon, A., and Capkun, S. (2012). Analysis of the communication between colluding applications on modern smartphones. In *ACSAC '12*.

Mazloom, S., Rezaeirad, M., Hunter, A., and McCoy, D. (2016). A security analysis of an in-vehicle infotainment and app platform. In *WOOT*.

Mercedes-Benz (2018). Mercedes-benz user experience: Revolution in the cockpit. https://www.mercedes-benz.com/en/mercedes-benz/innovation/mbux-mercedes-benz-user-experience-revolution-in-the-cockpit/.

Micinski, K., Phelps, P., and Foster, J. S. (2013). An empirical study of location truncation on android. *Weather*, 2:21.

Miller, C. and Valasek, C. (2015). Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

MirrorLink (2009). Mirrorlink. https://mirrorlink.com/.

MITRE (2013). CVE-2013-4787. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4787.

Nauman, M., Khan, S., and Zhang, X. (2010). Apex: extending android permission model and enforcement with user-defined runtime constraints. In *ASIACCS '10*.

Porter Felt, Chin, E., Hanna, S., Song, D., and Wagner, D. A. (2011a). Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638. ACM.

Porter Felt, Egelman, S., Finifter, M., Akhawe, D., Wagner, D., et al. (2012a). How to ask for permission. In *HotSec*.

Porter Felt, Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. (2011b). Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*.

Porter Felt, A., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. A. (2012b). Android permissions: user attention, comprehension, and behavior. In *SOUPS*, page 3. ACM.

Razaghpanah, A., Niaki, A. A., Vallina-Rodriguez, N., Sundaresan, S., Amann, J., and Gill, P. (2017). Studying tls usage in android apps. In *CoNEXT '17*.

Renault–Nissan Alliance (2018). Renault-nissan-mitsubishi and google join forces on next-generation infotainment. https://www.alliance-2022.com/news/renault-nissan-mitsubishi-and-google-join-forces-on-next-generation-infotainment/.

Reyes, I., Wiesekera, P., Razaghpanah, A., Reardon, J., Vallina-Rodriguez, N., Egelman, S., and Kreibich, C. (2017). " is our children's apps learning?" automatically detecting coppa violations. In *ConPro'17*.

Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9).

Schlegel, R., Zhang, K., Zhou, X.-y., Intwala, M., Kapadia, A., and Wang, X. (2011). Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS '11*.

Singleton, M. (2018). Spotify is testing a driving mode feature. https://www.theverge.com/2017/7/7/15937284/spotify-driving-mode-feature-testing.

Spotify (2018). Privacy policy. https://www.spotify.com/us/legal/privacy-policy/.

Tencent Keen Security Lab (2018). New vehicle security research by keenlab: Experimental security assessment of bmw cars. https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

Volkswagen (2018). 2018 passat press kit. https://media.vw.com/en-us/press-kits/2018-passat-press-kit.

Volvo Car Group (2018). Volvo cars to embed google assistant, google play store and google maps in next-generation infotainment system. https://www.media.volvocars.com/global/en-gb/media/pressreleases/228639/volvo-cars-to-embed-google-assistant-google-play-store-and-google-maps-in-next-generation-infotainme.

Warren, C. (2018). Radio fm. https://play.google.com/store/apps/details?id=com.radio.fmradio&hl=en&reviewId=gp%3AAOqpTOFWacIVZQ-JHULA86lKu5ZYSNQdIjsM8e6Ph0aj2RWN2aVmoFJFfmJhC91yQEErw6Z0Re3I0LF6k1V_o_Y.

APPENDIX

Table 3: List of attacks and their severity score, based on CVSS v3 (FIRST.Org Inc., 2018).

Name	CVSS v3 Vector	Score
SoundBlast	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:L	4.4
Fork bomb	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Intent storm	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9
Permissionless speed	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Permissionless exfiltration	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3
Covert channel	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3