FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Personengestüzte Informationengewinning durch Bluetooth Low Energy auf Android

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Gabor Zoltan Szivos, BSc
Matrikelnummer 01227443

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Dr.techn. Georg Merzdovnik

Wien, 22. August 2019

Gabor Zoltan Szivos　　Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Human-assisted information extraction through Bluetooth Low Energy on Android

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Medical Informatics

by

Gabor Zoltan Szivos, BSc
Registration Number 01227443

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Georg Merzdovnik

Vienna, 22nd August, 2019

Gabor Zoltan Szivos   Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Gabor Zoltan Szivos, BSc
Fred-Zinnemann-Platz 4/4/65, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. August 2019

Gabor Zoltan Szivos

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Sicherheitsprüfung der Firmware von Embedded-Geräten ist wichtig, um das höchste Sicherheitsniveau für die BenutzerInnen leisten zu können. Die Sicherheitsprüfung hat eine besondere Bedeutung für Geräte, deren Aufgabe das Sammeln von Medizindaten ist, oder die implantiert werden, um beispielsweise die Herzfrequenz zu regulieren, Medikamente automatisch zu verabreichen usw. In den meisten Fällen jedoch haben solche Geräte eine closed-source Firmware. Für weitere Analysen muss diese erst aus dem Gerät extrahiert werden, und dieser Vorgang verursacht bestimmte Schwierigkeiten.

Eine sehr verbreitete Methode um Updates für Embedded-Geräte zu verteilen ist heutzutage, sie mit einem Smartphone zu verbinden und durch Applikationen auf diesem Smartphone zu betreiben. In dieser Arbeit wurde ein System beschrieben, das im Stande ist, solche Applikationen zu analysieren und aus ihnen relevante Daten zu extrahieren, um einen Bluetooth-Low-Energy-Server aufzusetzen. Dieser Server kann dazu verwendet werden, um ein echtes Gerät zu ersetzen. Mit dieser Methode kann man sowohl das Protokoll, das für die Kommunikation zwischen Handy und Gerät verwendet wird analysieren, als auch die gesendeten Daten, wie ein Firmwareupdate, auslesen. Auf diese Weise wird einerseits das echte Gerät überflüssig, anderseits kann auf die komplizierte Firmware-Extrahieren verzichtet werden.

Zusätzlich wird im Laufe der vorliegenden Arbeit über die aktuellsten Methoden in Bereichen Firmwareanalyse, Sicherheit und Datenschutz von Embedded-Geräten und Android Applikationsanalyse berichtet.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Analyzing the security of the firmware of embedded devices is critical to provide the highest degree of safety and privacy for the user. This is true especially if the device is used to collect medical rated data, or is implanted into someone to regulate the heart rate of the patient, or to administrate drugs automatically. But most of such devices have a closed-source firmware, which can only be extracted from the device itself, which has its own difficulties.

Nowadays a widely used method to provide updates for consumer grade embedded devices is to connect them to a smartphone and operate them through an application on the phone. This thesis describes a system which is capable to analyze such smartphone applications, and extract relevant data from them to set up a generic Bluetooth Low Energy server which can pretend to be a real device. This makes it possible to analyze both the protocol used for the communication between device and phone as well as to dump any data which is sent from the phone to de device, like a firmware update. This eliminates both the need to have the device in question as well as the cumbersome firmware extraction.

This thesis also provides a thorough literature review in the fields of firmware analysis techniques and frameworks, the security and privacy of embedded devices and techniques and tools used to analyze Android applications.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Aim of Work 3
1.4 Methodological Approach 3
1.5 Structure of Work 4

2 Background **5**
2.1 Bluetooth 5
2.2 Android 11
2.3 Binary Analysis 15
2.4 Manual Reverse Engineering of APKs 17

3 State of the Art **21**
3.1 Firmware Analysis 21
3.2 Security and Privacy in the Field of IoT 23
3.3 Android Application Analysis 24

4 Implementation **31**
4.1 Generic GATT Server 31
4.2 Extending TaintDroid 32
4.3 Aggregating Results 35
4.4 Final Architecture 36

5 Evaluation **39**
5.1 Sample Set 39
5.2 Analysis Results 40
5.3 Result correlation 61

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6 **Discussion and Results** **65**
6.1 Comparison to Existing Solutions 65
6.2 Limitations 66
6.3 Discussion 69

7 **Summary and Future Work** **71**

8 **Appendix** **73**

List of Figures **75**

List of Tables **77**

List of Listings **79**

Bibliography **81**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

This chapter describes the motivation for writing this thesis, and discusses the problem regarding smartphone applications and smart device framework analysis. Afterwards, the goals required to solve the described problem are specified along with the methodological approach to meet these goals. Last but not least, the structure of the remaining part of the thesis is laid out shortly.

1.1 Motivation

The idea of automating time consuming or tiresome tasks has long been thriving humanity. Often these tasks include reading out the measured value of a sensor and act depending on the result.

For example: due to an illness a patient needs to go to the doctor's office every day, lets run some tests and measurements, and in case the results exceed some thresholds s/he should be given some medicaments, which for the sake of this example has to be prescribed by the doctor. The first step to automate this procedure might be to implant the measuring device into the patient to save time for both. However, it still requires the patient to go to the doctor, and let's say most of the time the threshold is not exceeded. The next step would be to integrate some connectivity modules into the implant which automatically sends the values to another node that decides whether or not an action should be taken, and if so, it sends a notification to the patient to see the doctor.

However, the firmware of such hardware was originally not designed to be connected to other devices or the Internet and so some design flaws which used to be untriggerable (because of physical inaccessibility) now might become exploitable.

This thesis summarizes the risks of such connectivity and gives a tool to help identify such hidden vulnerabilities.

1.2 Problem Statement

In the last couple of years the number of connected devices - also called smart device or "Internet of Things" device (IoT) - has steadily increased and presumably it will be over 31 billion by 2020. [1] This growth is mostly due to the fact that the price of connectivity modules, like WiFi- and Bluetooth-chips has significantly decreased, allowing consumer grade devices to be equipped with them.

The main issue regarding such connected devices is that because of the market pressure and the short time to market (TTM) security vulnerabilities are often included. [2] And despite serious certification processes [3], the same can happen to medical devices as well. The U.S. Food & Drug Administration (FDA) has reported that some selected pacemakers need to be updated due to security concerns. [4]

While such vulnerabilities are not new, due to the increased connectivity attackers can now exploit such vulnerabilities remotely and cause harm or death to people. [1]

As the software running these devices is often closed-source, the only way for independent security researchers to analyze them is to reverse engineer the firmware itself. The nature of embedded firmware poses specific problems to analysis, like interaction with additional hardware, lack of standardized distribution forms or limited access to process internals. These problems introduce complex challenges for automated analysis, as depending on the vendor of the device different approaches need to be applied. Furthermore, obfuscation might be applied to make the analysis more challenging.

Currently a widely accepted practice is that the IoT device, especially wearable devices like fitness-trackers, step-counters or heart rate sensors, do not communicate directly with the backend servers, but use the smartphone as a middle man. This is beneficial because of multiple reasons:

- using Bluetooth Low Energy (BLE) allows longer battery life and usage of smaller batteries
- syncing is not restricted to WiFi networks, as the device can utilize the smartphones data connection

Analyzing the smartphone counterpart of a smart device is easier because of many reasons. First and foremost, smartphone applications are easily and freely available, as they are hosted in application stores like Google Play, and not on the backend of the manufacturer like the firmware images of the devices. Secondly, smartphone applications - at least Android applications, which this thesis focuses on - are mainly written in Java, which is compiled into bytecode. It is relatively easy to decompile, not like binary images, for example a firmware. However, Android applications are often heavily obfuscated which makes manual analysis cumbersome, even for middle sized applications.

[1] Although up until 2015 no injuries or deaths have been reported from such intrusions. [5]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

1.3 Aim of Work

The aim of this work is to provide a human-assisted, semi-automatic framework for analysis of the communication between the smartphone application and the smart device. It includes static and dynamic analysis of Android applications to identify and pinpoint places where communication with the device counterpart is initiated.

In particular, the focus of this thesis will be put on the following topics:

- **An analysis of existing methods for Android application analysis:** This will provide an overview of the current state of the art and serve as a starting point for future work.
- **Requirement analysis for a semi-automated "application to device" communication analysis framework:** Based on the evaluation of existing analysis platforms, we will provide an in-depth analysis and taxonomy of concepts and problems that arise during analysis of Android applications.
- **Analysis framework for human-assisted communication analysis:** The identified requirements will be developed into an analysis framework to support analysts. It includes extracting data sent to and from the device, pinpointing places inside the application where communication is initiated and attempting to simulate the presence of the real smart device.
- **Evaluation of the framework and comparison to existing systems:** Based on the state of the art analysis and the selected Android applications we will provide a comparison of previous analysis approaches as well as our newly developed framework.

1.4 Methodological Approach

The methodological approach consists of the following parts:

1. **Literature review:** A comprehensive review of existing methodologies and technologies for analyzing Android applications, like taint tracking and Bluetooth communication analysis. It includes finding and analyzing existing frameworks which perform similar tasks.
2. **Sample set creation:** Search, collect and create Android applications which can be used for testing and comparing results. This is a very important task, as these example applications are going to be used during the implementation and testing phase of the framework. This set should represent the variety of ways and application can connect and communicate with a smart device.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. **Development of an improved application to device connection analysis framework:** In the context of evaluation possible optimizations and coverage for Android application analyses and extensions for existing tools and alternative approaches will be designed and developed.

4. **Prototype implementation:** Based on the findings and results of the previous steps a prototype will be developed, which will be able to conduct the required analysis tasks.

5. **Framework evaluation:** In this step a set of Android applications will be supplied for the developed framework and to other existing tools. The results of these different systems will be compared and evaluated to find possibilities for improvement.

1.5 Structure of Work

The rest of the thesis structures as follows:

- Chapter 2 describes concepts and technologies related to the remaining part of the thesis.
- Chapter 3 describes the current state of the art of the Android applications and binary analysis methods.
- Chapter 4 explains how the framework has been implemented.
- Chapter 5 evaluates the performance of the framework with the help of real-life fitness applications.
- Chapter 6 describes the obstacles during the implementation as well as compares the implemented framework to existing solutions and discusses the results of the evaluation.
- Chapter 7 concludes the findings of this thesis and lays out topics which will be addressed in the future.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 2

Background

In this chapter we are going to present some background information needed to understand the concepts and ideas in the following chapters. Firstly, we summarize the inner workings of the Bluetooth classic and Bluetooth Low energy technology. Secondly we dive into the system architecture and the layers of the Android operating system which is currently one of the most widespread mobile operating systems. Next, we present various binary analysis techniques, and their respective advantages and disadvantages. Lastly, we describe the steps taken during the analysis of the applications written for the Android OS.

2.1 Bluetooth

According to the Bluetooth Special Interest Group (SIG) - the creator and maintainer of the Bluetooth Standard - almost 4 billion Bluetooth devices were expected to be shipped only in 2018. [6] It makes Bluetooth one of the most widespread connectivity standards in the world.

The Bluetooth Standard was first published in 2002 as part of the IEEE 802.15 working group. Until 2010 only one form of Bluetooth existed which was called Basic Rate (BR). It was later extended with upgraded data transfer options and was called Enhanced Data Rate (EDR). This operation mode is also called Bluetooth Classic nowadays. In the 4.0 version of Bluetooth, which was released in 2010, SIG released the specification for Bluetooth Low Energy (BLE), which promised cheaper components and lower energy consumption.

2.1.1 Bluetooth Stack

The Bluetooth specification defines the core system as a host and one or more controllers. The host and the controller(s) communicate through the Host Controller Interface (HCI).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The controller is the logical representation of the components under the HCI, whereas the host represents everything above the HCI. The specification defines three types of controllers with the first two being primary while the last one being secondary.

- BR/EDR controller
- LE controller
- Alternate MAC/PHY (AMP) controller

Figure 2.1 shows the core Bluetooth stack, which will be described in the next few subsections.

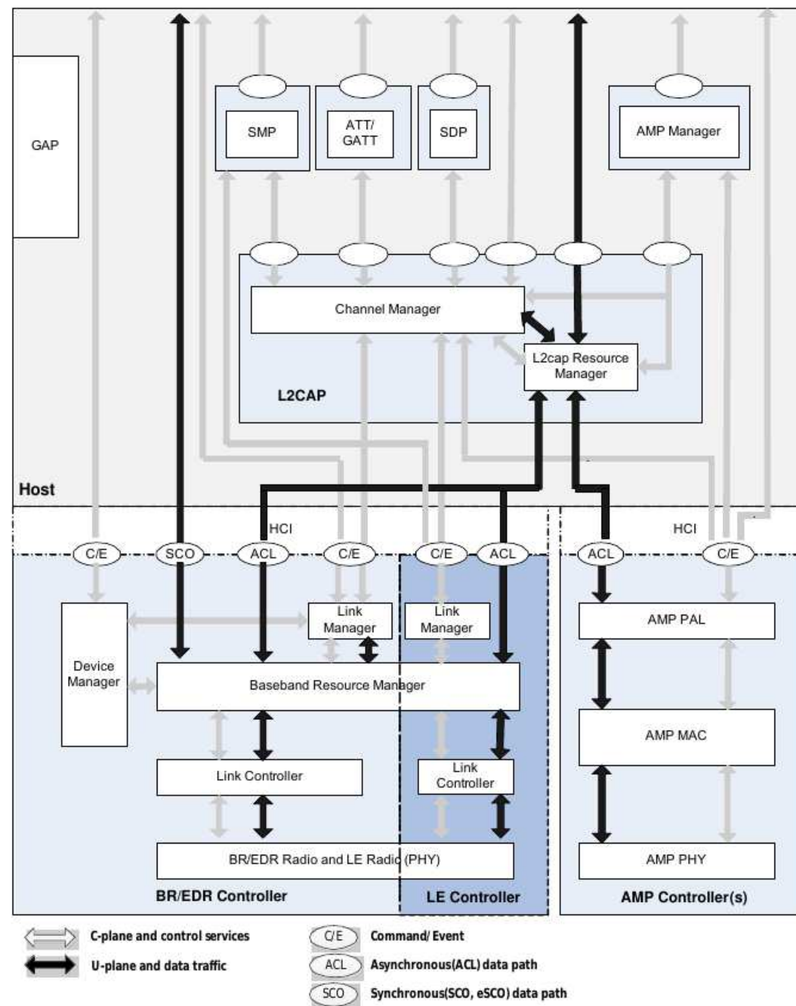
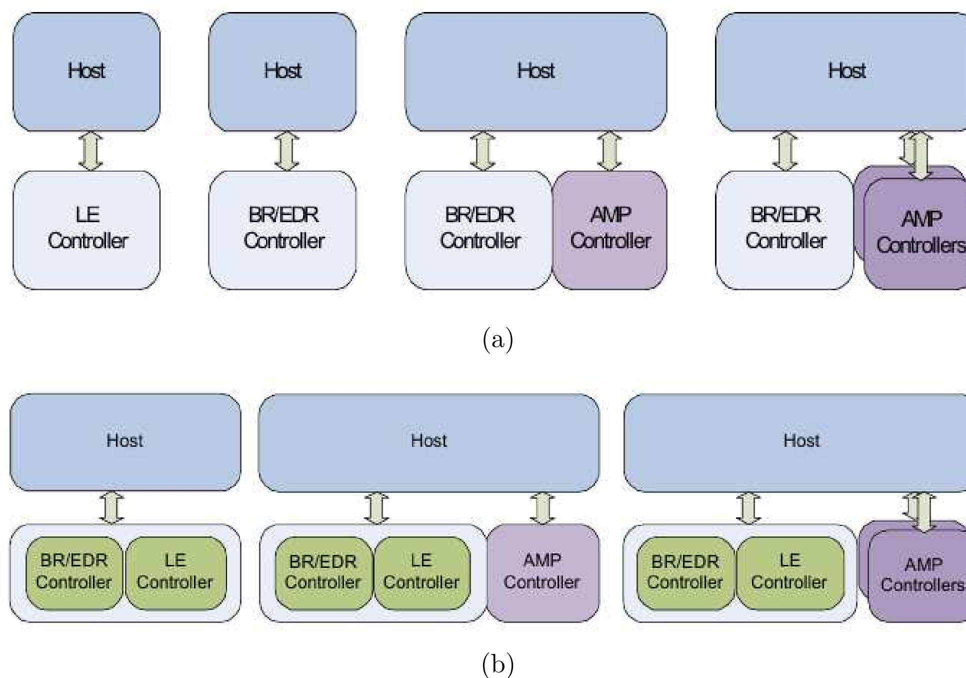


Figure 2.1: Bluetooth-stack [7]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As one device can include several controllers, there are multiple ways how controllers can be combined, only providing exclusively BR/EDR or LE or a combination of these two with or without one or more Alternate MAC/PHY (AMP) controllers. It is worth to mention that although multiple secondary controllers can be found in a device, the specification states that only one primary controller can exist. See Figure 2.2a and 2.2b for the visual representation.



(a)

(b)

Figure 2.2: Possible controller configurations[7]

2.1.2 Bluetooth Classic

Devices which connect through Bluetooth form piconets. Inside this network there is a device called the master and every other device is a slave. A device can participate in multiple piconets and is allowed to have the master role in one and be a slave in another one. The master device provides the clock and the frequency hopping pattern for the network. The slaves synchronize themselves to the master. A minimal BR/EDR controller should have the following components which are going to be described in detail later in this subsection:

- Radio
- Baseband Resource Manager
- Link Manager

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Host Controller Interface (HCI)

Physical Layer or also called Radio Layer is responsible for the transmission of the assembled Bluetooth packages over the unlicensed 2.4 GHz radio band. This radio frequency band is divided into 79 channels which are separated by 1 MHz. To reduce possible interference between devices frequency hopping is applied. It means that at certain times the devices which are connected to each other change the currently used channel to another one. The default rate of frequency hopping is 1600 hops every second. To reduce interference even more, optionally Adaptive Frequency Hopping (AFH) can be used. This means the currently used channels are detected through active measurements and are excluded from the list of usable channels.

Baseband Resource Manager grants access to the physical layer. The responsibilities of the Baseband Resource Manager are twofold. On the one hand it schedules entities to get access to the physical layer for the specific time they have negotiated. On the other hand, it negotiates with new entities for access time in the form of contracts. These contracts represent the commitment of the manager that the specific quality of service will be met to guarantee the expected performance for the higher level applications.

Link Manager as the name suggests manages logical links between devices. It includes the creation, modification and release of these links, as well as the modification of the physical link properties of the connected devices. For this the so called Link Manager Protocol is used.

The following operations can be performed by a BR/EDR device, which is initiated by the Link Manager:

Inquiry
: A scanning device broadcasts inquiry requests, and the devices which are visible to other devices answer with a response.

Paging
: This is also called connecting. One device sends a page request to a special channel which is used by the connectable device for listening to connection requests.

Role Switching
: Switching master and slave roles.

Host Controller Interface provides an interface through which the functionality of the controllers can be accessed, regardless of the type of the controller.

2.1.3 BLE

Bluetooth Low Energy shares some similarities to Bluetooth Classic. As Figure 2.1 shows BLE controllers have similarly a physical layer - also called as “LE PHY” -, which also operates in the same unlicensed 2.4 GHz ISM band, however with only 40 channels which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

are separated by 2 MHz. From these 40 channels three are used for primary advertising and the rest for secondary advertising and data transfer.

One significant difference to the Link Manager of BLE is that, unlike the Bluetooth Classic controller, it uses the Link Layer Protocol for managing logical links between devices and not the Link Manager Protocol. Furthermore, the Link Manager of BLE performs different operations than the Bluetooth Classic counterpart:

Advertising An advertiser sends and an advertising package on one or more primary advertising channels. The advertiser can send additional data packages as well, if it decides to, which may or may not be sent on the primary channels.

Scanning A scanner can either scan passively, that is the scanner only listens to and processes advertising packages, or scan actively when receiving an advertising package it sends an additional scanning package to the advertiser to learn more about it. [8]

Connecting A scanner upon receiving a connectable advertising package can send a connection request, effectively changing into the role of an initiator.

Although connected devices form a piconet, it also has some differences compared to Bluetooth Classic. First of all, each slave in a piconet has its own data channel, whereas in a Bluetooth Classic piconet it is shared with every slave. Although the classical master and slave roles are present in a piconet, there are additional types of actors defined for BLE. A device which periodically sends out so called advertising packages on one of the advertising channels is called an *advertiser*. A *scanner* is a device, which listens to advertising packages on the advertising channels, without being connected to the advertiser. And last but not least an *initiator* is a device which initiates a connection to the advertiser. Having successfully connected to the advertiser the initiator becomes the slave and the advertiser the master, in this way a piconet is formed.

2.1.4 AMP

The secondary controller type called AMP is only mentioned for the sake of completeness as describing it in detail would go beyond the scope of this thesis.

After connecting through BR/EDR, the AMP manager can discover AMP capabilities on the other device and transfer the data transmission to use the AMP interface in order to reach significantly higher transfer speeds.

2.1.5 Logical Link Control and Adaptation Protocol (L2CAP)

The L2CAP handles channel and protocol multiplexing, segmentation and reassembly (SAR) of packages, flow control for every channel and error handling. It makes possible for the various controllers (BR/EDR, LE or AMP) to interact with higher level protocols, like Attribute Protocol (ATT)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The Attribute Protocol runs through a dedicated L2CAP channel during the communication between the attribute server and the client. The client is able to send requests, commands and confirmations, to which the server answers with notifications, responses and indications. The requests and commands enable the client to read out or write to an attribute of the server.

2.1.6 Bluetooth Profiles

Bluetooth profiles define what kind of feature requirements the device or the application has to make communication with other devices or applications possible. Moreover, they also define data formats used by the application/device and its behavior.

Generic Access Profile (GAP) is a profile that every device must have, as it provides its basic system definition. In case of BLE the GAP would require the device to have the predefined controller elements (PHY, the Link Layer), on the host side the L2CAP module, a security manager for encrypting the traffic and both support for both ATT and the Generic Attribute Profile (GATT). Furthermore, it defines the LE-GAP different roles a device can have:

Broadcaster the application using this role only broadcasts data, but does not accept connections

Observer the application only receives broadcasts, and like the Broadcaster, does not support connections

Peripheral is used by devices which support a single connection and never take the master role

Central is a device that is able to connect to more than one peripheral devices

GATT is a protocol which must be implemented in case of BLE. As GATT builds on top of ATT, there is a GATT server and a GATT client. GATT also defines that the data should be stored as Services and Characteristics on the server. A server can have one or more services, which again can include multiple characteristics, whereas the characteristics can include any number of descriptors, which describe the data stored in a characteristic.

A service can be either primary or secondary. Primary services provide the main functionality of the device. Secondary services provide additional functionality, and must be referenced by at least one primary service. Referencing a service means that every element of the referenced service becomes a part of the referee as if it were its own, during which the referenced service still exists as an individual one.

A characteristic is nothing else than a value stored on the server. As stated before it can have multiple descriptors, which can provide a context for the user what that value means and where it is used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

A GATT profile defines which devices should have which roles and which services the device should provide. As an example the Health Thermometer Profile (HTP) states that the thermometer itself should take the server's role and the collector the client's role. The server should include the Device Information Service and the Health Thermometer Service. [9] The Device Information Service is defined as a primary service which has the 0x180A UUID, and includes multiple characteristics like the "Manufacturer Name String" or the "Serial Number String" which have predefined UUIDs. [10] The Health Thermometer Service, on the other hand, has the UUID 0x1809 and has characteristics like "Temperature Measurement" or "Measurement Interval". [11]

2.2 Android

Like Figure 2.3 shows, Android is one of the most widespread operating systems in the mobile device world. It started off as an operation system for phones, but nowadays it can be deployed on tablets, IoT devices, like smart watches, laptops, like Chromebooks or even in car infotainment systems.

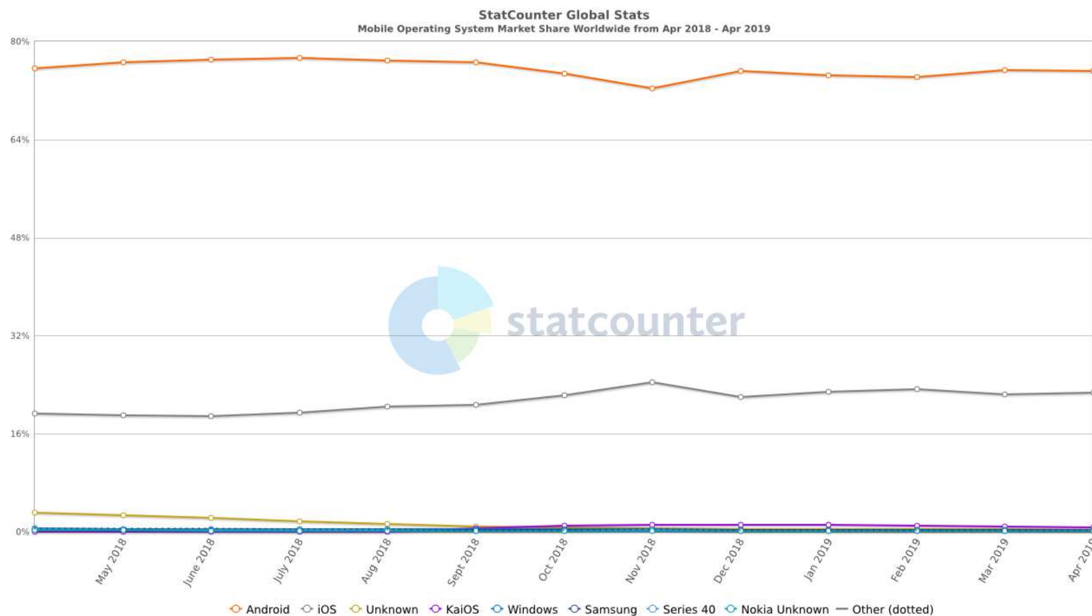


Figure 2.3: Worldwide mobile OS market share [12]

In the next sections each element of the Android software stack, see Figure 2.4, will be described.

2.2.1 Android Stack

Android is built on the Linux Kernel which is extended with the SELinux module to provide additional security features. On the top of that is the Hardware Abstraction

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

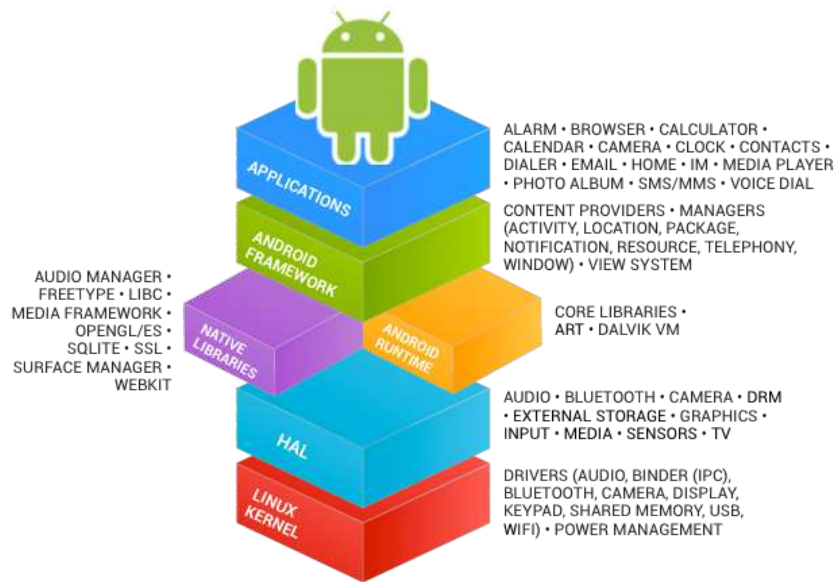


Figure 2.4: Android-stack [13]

Layer, which allows seamless interaction with the actual device drivers without changing the higher levels of the stack. The next layer includes native libraries like LibC or SSL-Implementations like OpenSSL, and the core libraries along with the runtime environment of Android, which up until Android 4.4 was the Dalvik Virtual Machine (DVM) and later it was changed to the Android Runtime (ART). Between any application and the runtime is the Application Framework which provides various utilities for applications to interact with the phone itself.

Kernel As Maker et. al. describe, although Android is built on the Linux kernel there are some differences. For one Android includes additional modules, like the alarm driver, or the binder driver just to name a few. [14]

As mentioned before SELinux is turned on by default on all Android devices to enforce a strong security policy. The core idea of SELinux is to label every entity in Linux, like files, processes, users and so on. These labels are used by the security policy to decide what a particular entity with a specific label can do. Another concept of SELinux is that everything is prohibited that is not explicitly allowed. [15]

Furthermore, the kernel includes the actual device drivers either as kernel modules or being compiled into the kernel itself.

HAL Yaghmour describes the HAL as a hardware library loader, as it consists of several shared libraries which are implemented by either the manufacturer or Android itself. Through these libraries higher layers of Android can interact with the hardware without using directly the kernel itself. [16]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure 2.5 visualizes how the HAL is built up.

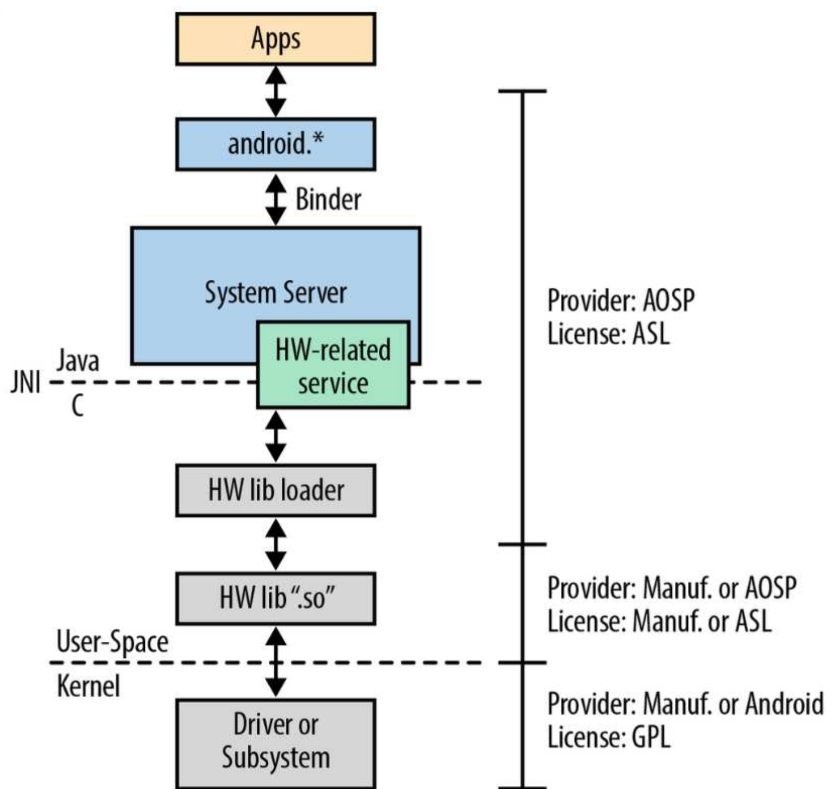


Figure 2.5: Layers of the Hardware Abstraction Layer [16]

Android Runtime There are two runtime systems on Android. The newer one is called Android Runtime (ART) and was introduced with Android 4.4 as experimental, and with Android 5.0 it replaced the older one completely. As mentioned before the older runtime is called the Dalvik Virtual Machine (DVM), which is very similar to the Java VM (JVM).

One major difference between the JVM and DVM is that JVM uses the class-files - the output of the Java compiler - directly, while for DVM these class-files are converted into a new file which is called *Dalvik Executable* (dex). The main idea behind dex-files is to eliminate duplicates within multiple classes, and so reduce the memory needed for applications and libraries. [17]

During the booting of the device a process called Zygote is started in the DVM which loads core libraries and after having started it waits for further commands. As soon as an application starts, the Zygote forks, and the child process inherits the loaded libraries and becomes the application. In this way the starting time of an application can be reduced as often used libraries are already loaded by the Zygote. [18]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Dalvik also houses a Just-in-Time (JIT) compiler, which gets executed as the application is started. With JIT the byte-code is turned into native binaries which, on the one hand, makes loading the application longer, on the other hand, as it runs, everything gets faster, as the code is not executed through the VM anymore. [16]

Although this thesis mainly concerns with the DVM, for sake of completeness the ART will be discussed shortly as well. The main difference to DVM is that ART uses Ahead-of-Time (AOT) compilation with the combination of the JIT. Basically the idea is that the byte-code of the installed application is not compiled right away into native executable but either it gets JITed or it is executed by an interpreter, but in both cases the most frequently used parts of the application are marked. Later as the device is idle these frequently used parts get compiled into native code by the AOT compiler. As the application is restarted the AOT compiled parts are executed as native code without the need of JIT, and the analysis continues, so on demand more and more parts of the application gets AOTed. [19]

Native Libraries Some system functions are implemented as native libraries. They are called native, as these libraries are implemented in C or C++ and compiled specifically for the CPU-architecture of the device. The application framework, or also called as Android framework can call these libraries through the Java Native Interface (JNI). Native libraries can exist on the application layer as well as application developers have the right to implement functionality in compiled languages.

Native system libraries include for example Bionic which is the LibC used for Android, or a lightweight SQLite-engine, as SQLite is used for persistent storage by the applications. [17]

Android Framework The last layer beneath the application layer is the Android Framework or Application Framework. This layer provides several services which makes it easy for the developers to access system functions through a defined API without using the native libraries directly. For example: if the developed application were able to read or send SMSs and MMSs or provide calling functionality, the Telephony Manager would be called. If the application were to access the GPS sensors or were to scan the available WiFi-networks, the Location Manager would be the right entity to ask. [17]

Applications Although until now we have referred to an application as it was one single entity, in fact consists of multiple components. There are four types of application components, and a single application can include any number of those. *Activities* are components which provide a graphical user interface, and thus are used to interact with the user. An activity can be as simple as some static text on a white background, or as complex as the UI of a mobile game. *Services* are background running tasks which do not provide any user interface, and are meant not to interact with the user directly. *Broadcast receivers* listen to specific broadcasts, and upon receiving the broadcast they trigger an event. A broadcast receiver can for example listen to state changes of the

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bluetooth interface, like turning on or off. Lastly, a *content provider* represents an entity which manages the data of an application. This could abstract anything from an SQL database to a backend server on the Internet. [20]

2.3 Binary Analysis

As firmware analysis, which is discussed in Chapter 3, is a specialized form of binary analysis, this section provides an overview about the core techniques and approaches used during binary analysis.

2.3.1 Static Binary Analysis

The goal of static analysis is to gather as much information about the binary object as possible without ever starting it. The advantage of this approach is, that setting up the proper runtime environment is not needed to conduct the analysis. This advantage is very handy in cases where running the binary has special needs regarding its environment. These needs can be anything from finding a specific file at a specific directory on the file system through requiring additional, in some circumstances even proprietary and expensive software to cases where dedicated hardware is needed on which the binary can run.

Apart from obvious techniques like extracting strings and files packed into the binary, there are two main methods which can provide insight into the inner workings of a binary executable, which are disassembly and decompilation of the executable.

Disassembly

Disassembly of binaries is a technique, where the binary is converted into a more human-readable format represented by the assembly code. Depending on the architecture this process can get more or less complicated. Binaries, compiled for CPU architectures like ARM, where the length of an instruction is well defined and set to either 4 bytes (in ARM state) or 2 bytes (in Thumb state), can be disassembled easier, as the disassembler does not have to decide how many bytes belong to the next instruction, thus it will less likely be tricked. Whereas binaries for architectures with a variable instruction length, like x86, can confuse the disassembler easier in some cases.

The two different methods, which can be used by disassemblers to disassemble binaries with variable length instructions, are *linear sweep* and *recursive traversal*.

Linear Sweep disassemblers take one instruction after another starting at the beginning of the segment containing instructions. They are not very complicated to implement and work efficiently, however, they can be confused in cases where data is mixed in-between instructions, which can lead to a desynchronized state outputting incorrect instructions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Recursive Traversal disassemblers, on the other hand, use additional information found in the binary. They follow the control flow of the binary, like calling a function, or following a jump, and only return to the call-site after that branch has been fully explored. However, as soon as addresses used by jumps are generated at runtime, also called as indirect jumps, this strategy cannot provide any additional information and has to revert to linear sweep.

Decompilation

Although assembly is considered to be a completely human-readable language, the nature of low level languages and the big amount of code generated can be very daunting. The goal of decompilation is to generate higher level language representation of the binary from the disassembly (or other intermediate representation of the executable). The main difficulty, at least for compiled languages like C, at decompilation is, that the compiler discards information which is not valuable for the CPU itself, like types or variable names. Additionally one instruction on the higher level usually maps to multiple instructions on the machine level. The combination of these two properties make it really hard to create a correct mapping in the other direction.

2.3.2 Dynamic Binary Analysis

The complement of static analysis is dynamic analysis, during which the binary is executed and observed to gather information during runtime. It can overcome difficulties static analysis faces, like inspecting dynamically generated contents, self-modification during runtime or determining the target of indirect jumps. The disadvantage is however, that only the currently executing path can be observed, so it is really hard to gather information about the whole binary. And, as mentioned before, the binary might need a special environment to run.

In the following some of the most widespread dynamic analysis techniques will be described.

Symbolic Execution

It is not uncommon that an executable does different things depending on the inputs it was fed, or what options it was started. The main idea behind the symbolic execution is to execute the binary *"virtually"* instead of running it manually or with randomly chosen inputs or starting options. Virtually in this case means that every input variable is changed to a symbolic one, which can have any possible value. As soon as a decision is to be made which path the execution should take based on the value of a symbolic variable, both paths are followed and so more execution paths can be explored. [21]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Taint Analysis

The idea behind taint tracking or also called as information-flow tracking is that pieces of relevant information, like user input, or sensitive data is observed during execution to see where it lands. Information gets tainted at the so called *sources*, and a notification is generated if tainted information arrives at a *sink*. The *taint policy* defines what kind of information gets tainted in which cases and where the taint gets checked. The policy specifies the strategy for taint propagation. Taint must be propagated if the new variable or information was effected either explicitly or implicitly by another already tainted data.

Explicit data flow is where tainted data is directly involved in the creation of a new value, whereas implicit data flow is where the tainted data has an affect on the control flow of the program, and thus the value of untainted data is effected. [22]

If this policy is poorly chosen either the framework will either *overtaint* values, which means data gets tainted which is neither comes from a source nor is it derived directly or indirectly from other data that has already been tainted, or *undertaint* values, which means data which should have been tainted will not be. [23]

Although taint analysis is listed under dynamic analysis techniques, there are static information-flow analysis methods. Static analysis can either be built into the language itself [24] or can operate on existing applications. In the latter case it usually generates some sort of graph in which data-flow can be modeled and tracked. [25, 26, 27, 28, 29, 30] One major drawback of static analysis is that, as the program is not executed, information which is generated during runtime cannot be analyzed. One such case would be reflection, in which case with the help of dynamically generated strings an existing function is called. Dynamic analysis tools, on the contrary, operate with the help of instrumentation. It means the tainting logic is built into the instrumented runtime environment effectively extending runtime functionality with applying and checking taint. [31, 32]

Fuzzing

Fuzzing is a technique which can be used to identify bugs based on receiving unexpected data or data in an unexpected format. The fuzzer sends a specific input to the firmware or the program under analysis, and in the next round the input is mutated. Mutation in this context means that the input gets slightly changed. Every crash of the program is reported as it potentially means, that there is some sort of an unhandled case which might mean that it is exploitable.

2.4 Manual Reverse Engineering of APKs

The first step to analyze any application is to acquire a copy of it. In case of Android the main source of applications is the official application repository, the Play Store provided by Google . Unfortunately, Google does not provide any API to interact with the repository to search for or download applications directly. In order to get the application files the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

app needs to be installed on a phone. After the installation with the help of the official debug interface for Android phones the APK can be searched for and downloaded from the phone. Listing 2.1 shows how we can search for the APK file of a given application and with the adb pull command the APK file can be downloaded from the phone.

```
# Get the package name of the app we search for
APP_PATH=$(adb shell pm list package -f | grep <app name>)
# First stage of parsing -> remove unnecessary prefix
APP_PATH=$(echo $APP_PATH | awk -F':' '{print $2}')
# Second stage of parsing -> remove unnecessary postfix
echo $APP_PATH  | awk -F'.apk' '{print $1".apk"}'
```

Listing 2.1: Search for an application on the phone

Alternatively, some APKs are hosted on third party mirror sites like apkmirror[1]. Through such sites often more versions of the same application are available.

As soon as the APK of the application is acquired, it can be disassembled with one of the tools mentioned in Section 3.3.1, like apktool. The result will be the decoded Manifest file, the disassembled classfiles in smali format and the decoded resources. Figure 2.6 shows what apktool does with an APK.

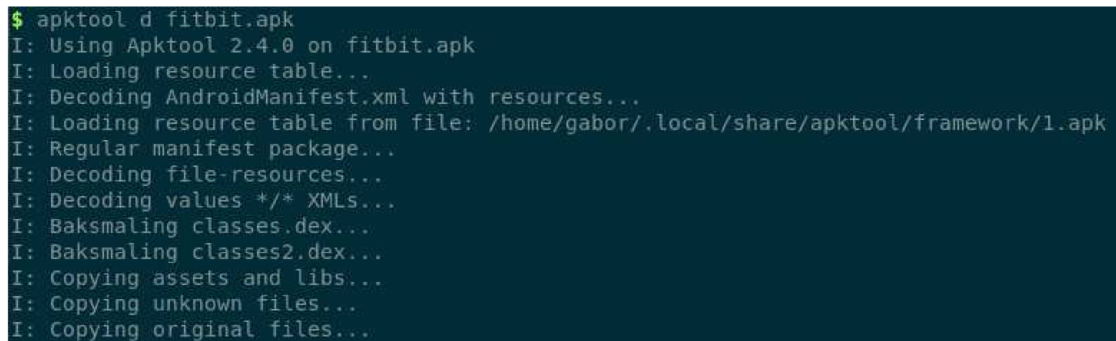


Figure 2.6: Output of apktool

At this point the actual analysis could be started, however, even a moderately big APK like the one used in the previously mentioned Figure includes around 13000 smali files. About half of these files are, however, libraries which were compiled into the application, and only in special circumstances are they important for the analysis of the application logic. The rest of the smali files have an average length of around 300 lines. Obviously, understanding such a big amount of disassembly without a specific goal is just unfeasible. If the great number of files and huge amount of code are not be enough, the applications are obfuscated in most cases, which only makes manual analysis using the disassembly even more cumbersome.

[1] https://apkmirror.com

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

However, as described later in Section 3.3.1, decompiling APKs is not that hard, which results in Java-code. At this point IDE-like environments like *jadx-gui* or the *bytecode viewer* can be utilized. They not only include the required tools like decompilers and disassemblers, but also provide functionality like text and code search or jump to object or method definitions.

Changing the decompiled application, building it again to test a hypothesis or adding functionality, will not work all the time, as in some cases the decompiler will not be able to decompile certain parts of the application which would result in compile errors during the building. If changing the application is absolutely necessary, it is advisable to apply the change directly in smali and re-build the APK from smali again using for example apktool.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

State of the Art

This chapter investigates the current state of the art research in the fields of static and dynamic binary analyses, as well as the security and privacy of connected embedded systems, and last but not least, various techniques to analyze Android applications.

3.1 Firmware Analysis

In the following sections different dynamic and static state of the art solutions - which are specialized to analyze firmwares - are going to be presented.

3.1.1 Dynamic Analysis

In the following section existing dynamic analysis frameworks are described.

It is not an easy task to dynamically analyze firmwares. Mainly because they need an environment provided by a special hardware. It means they will most likely try to access some special peripherals and expect a very defined behavior from them. Zaddach et. al. [33] describe three common ways of dynamically analyze firmwares which run on a special hardware, and thus expect a special behavior. *Complete hardware emulation* means that the needed hardware of the firmware in question is emulated completely, which works very well for documented and standardized systems, but it is almost impossible to achieve for undocumented complex and proprietary system-on-chips. *Hardware Over-Approximation* is a technique where the emulator models the hardware in a generic way where some properties are assumed, as an interruption can occur at any point of time. As it is described it works well for small systems, but one should count on false positives, since this technique on larger systems usually leads to state explosion. Last but not least *firmware adaptation* is where the firmware, or a part of it, is ported to another more generic emulator-environment. A fourth option is also proposed, which is a hybrid solution, where they execute part of the firmware directly on the hardware.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Although both firmalice [34] and the framework proposed by Costin et. al. [35] can mainly be categorized as dynamic analysis tools, in reality both of them utilize static techniques as well. The former uses static analysis to identify interesting parts in the firmware to save resources and time during the dynamic analysis, and the latter tries to find known bugs in PHP code and CGIs. The dynamic analysis is, however, different from the previously mentioned frameworks. Firmalice executes the framework dynamically with a custom symbolic execution engine, whereas Costin's framework starts the web server with QEMU and various web penetration tester tools.

FIRMADYNE [36], which is similar to Costin's framework, uses QEMU with custom kernels. However, to infer system and network settings instead of searching and parsing the unpacked file system, they initiate a pre-emulation, and the custom kernel reports back the needed information. According to the acquired information QEMU gets reconfigured and the real analysis is started.

Both Avatar [33] and SURROGATES [37] utilize hybrid dynamic analysis techniques. It means that instead of emulating or assuming peripheral behavior the system forwards memory access requests to the real hardware, whereas the hardware response is channeled back to the emulator achieving high precision regarding memory operations. The main difference is that Koscher et. al. implemented an FPGA-based PCI-Express JTAG adapter instead of a serial one like in Avatar, which enables near-real-time dynamic analysis.

On the contrary to the previously mentioned frameworks, FIE [38] requires the source code of the firmware. During Davidsons' analysis the source code is compiled into LLVM bytecode, which then gets executed by the KLEE [21] symbolic execution engine.

Subramanyan et. al. [39] propose a method to find confidentiality and integrity violations of critical information within firmwares with the help of symbolic execution. To achieve it they provide a modeling language for information flow properties which can be translated into symbolic variable constraints. Moreover, Subramanyan et. al. also provide an algorithm for verifying the absence of information flow violations.

To enhance the efficiency of the fuzzing, it can be guided by for example symbolic execution or taint-tracking. [40, 41, 42]

Directly comparing executables, even if they are just two versions of the same codebase, can be problematic, as compilers can reorganize the whole executable due to optimizations. If one tries to directly compare different executables, like a compression program and a web server, it is more or less impossible, as their codebase is so different. The previously mentioned problems make it hard to verify if a security patch really fixed the vulnerability, or if the same vulnerability could be found in different applications due to code reuse. One way to overcome it is extracting the control flow graph of the respective programs. The control flow graph (CFG) of a program is a directed graph where the nodes are considered to be basic blocks - a sequence of instructions with only one entry and exit point - and the edges are control flow paths. [43] One way to compare CFGs is to execute the basic blocks of two binaries symbolically and to apply a theorem solver to verify if

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the effects of the block is always the same. [44] On the other side Ming et. al. use taint analysis, so that the number of matching candidates can be reduced to speed up the process. [45]

3.1.2 Static Analysis

In the following section we describe existing static analysis frameworks designed to work with firmwares.

Costin et. al. [46] propose a framework which is capable of analyzing firmware images on a large scale. Instead of performing a code analysis, their main goal is to extract sensitive information, like RSA private keys and passwords, from firmware images by unpacking them. Besides the information extraction they try to identify packages with known vulnerabilities and dangerous misconfigurations, like running a web server with a privileged user.

Several frameworks [47, 48, 49, 50] extract the control flow graph of binaries, but instead of using it for decompilation they try to match it with other known control flow graph signatures in order to find known bugs and vulnerabilities. Control flow graphs can also be used to identify changes among different versions of the same executable. This way one can easily identify which parts of the binary have been patched. This also enables the propagation of previously created meta-information, like function names and other annotations, to a newer version of an executable to save time to redo parts of the analysis. [51, 52]

Thomas et. al. [53] utilize machine learning to find backdoors in the firmware of consumer off-the-shelf embedded devices. A trained classifier puts a class label on every executable found in the firmware in question. For each class of binaries there is a functionality profile which declares the expected functionality for this class. Within the scope of the static analysis of the binaries their functionality is compared with the functionality profile.

3.2 Security and Privacy in the Field of IoT

Embedded devices are usually designed to execute one specific task and not to be multipurpose like a laptop or PC. To make the production cost-effective their power-source, if they only operate with batteries, and the amount of memory they have is adjusted to the needs of their task. It naturally leads to hardware based limitations in the security domain. Security related tasks, like encryption, are usually computation intensive operations, which means such algorithms cannot be directly ported to an embedded device, due to the limited computational and battery power. Moreover, the limited volatile memory poses another problem, as encryption algorithms are not necessarily designed to be memory efficient. [54, 55]

In the field of implantable medical devices privacy and security is even more critical, as unauthorized access to the device can potentially lead to life threatening situations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

However, as Halperin et. al. [55] describe there are clear tensions between security and utility or safety goals. One such tension is how someone can access or deactivate an implanted device. It is clear, that for everyday use strong access control is desired, but in an emergency situation in which the patient may not be able to allow access, the implantable device might even hinder the healthcare professionals or even threaten them, for instance in case of an implantable cardiac defibrillator. Proposals for solving or at least reducing the aforementioned tensions would be for example to validate the emergency access by connecting the external programmer to the Internet, where a backend could decide whether the emergency access is justified or not. Another idea by Halperin et. al. is to be able to revoke privileges for stolen programmers with the help of for example a certificate infrastructure.

It is not unusual that the operation environment of a device is potentially hostile or at least it cannot be secured easily, because the device serves the public and it must be accessible for anybody, like for example an automated external defibrillator. For such devices it is crucial that a malicious actor cannot just upload his or her malicious firmware, which potentially could just break the device causing denial of service or implement malicious functions which would alter the expected behavior. Hanna et. al. [56] recommend the usage of cryptographically secure device updates, which means that the device should not only check a specific checksum to determine transfer errors but also verify the validity of the signature of the update file, which was created by a secure cryptographic algorithm.

Folk et. al. [57] address several more issues which need to be clarified in the field of IoT, like data ownership or operation issues from the point of view of the different actors. The main idea of the IoT-devices that they collect data and send them to be analyzed. However, as Folk et. al. describe the question arises whom this data belongs to, as this field is not well regulated, except for medical data, but again what about data which is not directly used for diagnosis or treatment, like the number of steps taken a day. Different actors have different expectations and issues regarding the operation which can be conflicting. For example the end-user and the system administrators require that the system in use should secure, however, supporting old devices is a cost issue for the manufacturer, whose goal is to rapidly develop new devices and make older devices relatively fast obsolete in order to increase sales. This, of course, leads to the conflict, that an older device will not get any updates after a certain time, which can be right after being introduced to the market, and it leaves devices vulnerable. As long as such issues are not regulated, these conflicts remain unsolved and mostly lead to the fact that the manufacturer chooses the most cost effective solution.

3.3 Android Application Analysis

Similar to binaries, Android applications can be analyzed both statically and dynamically. Some parts of the analysis are even easier for compiled languages, since Android applications are mostly written in Java, which gets compiled into bytecode which by its

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

nature must include a relatively great number of metadata.

In the following sections different methods and tools are shown to analyze Android applications.

3.3.1 Static Analysis

Pure static analysis on Android is not utterly complicated. There are around 230 different instructions, which are completely manageable. It makes the disassembly of the application also relatively straightforward. And additionally the generated bytecode includes additional sources of information, like class and method names, which makes the decompilation of the application considerably easier than for example in the case of compiled languages.

Apkanalyser [58] and **apktool** [59] are very similar tools. Both of them are able to retrieve various resources, like XML-files or images. They also include a disassembler which turns a Dex file into multiple smali files, where smali is an assembly-like representation of the Dalvik bytecode. Additionally, apktool can also rebuild an apk using the decoded resources, even if they are modified.

Dex2jar [60] is a tool which maps Dalvik bytecode to java bytecode. The former has several sub components:

dex-reader	reads files in the Dalvik Executable format
dex-translator	converts Dalvik instructions into `dex-ir` format
dex-ir	represents a Dalvik instruction
dex-tools	lets the user work with the Java class files, like modify the apk or decrypt the strings
d2j-smali	disassembles the dex file into smali files
dex-writer	acts as a dex-reader just for writing

Similarly dex2jar, **ded** [61] converts dex files to Java classes by “retargeting” the Dalvik bytecode to java bytecode. Additionally, ded optimizes the generated java bytecodes with Soot to make the results more user friendly. After having been generated, the class-files can be decompiled into Java source files with tools like the Java decompiler (jd) [62].

The android equivalent of jd is **jadx** [63]. It decompiles apk files directly into Java classes and offers “de-obfuscation” options, although the de-obfuscator of jadx only gives each variable, method, class and package a unique auto-generated name. The user can, however, supply a mapping file to use, in this case that one will be used. Jadx can also export an apk as a Gradle project.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Last but not least, **androguard** [64] is a framework implemented in Python to analyze Android applications. It features a disassembler for dex and apk files, a decompiler which is based on a control flow graph retrieval and abstract syntax trees. Furthermore, androguard is capable of decoding the binary XML format of Android and so recovering the manifest file of the application and the used resources. Androguard also supports the jadx decompiler, in case the built-in one did not work.

Tainting based static analysis methods will be described in Section 3.3.3.

3.3.2 De-obfuscation of Android Applications

Although obfuscation is often connected with malign programs, in the Android ecosystem obfuscated applications are not uncommon. This is because the Android SDK provides a tool called ProGuard, which is recommended to be used for releases. ProGuard not only obfuscates the application by changing its layout, but also minifies the application by removing unused code parts and resources.

DeGuard [65] applies probabilistic models to de-obfuscate Android applications. For the de-obfuscation DeGuard creates an annotated dependency graph, where nodes are program elements. Each node is annotated either as known (non-obfuscated) or unknown (obfuscated). An example for known program element would be something which is provided by Android itself like `TelephonyManager`. Such elements cannot be obfuscated, as otherwise the classloader would not find them. A set of constraints is generated for the application to prevent renames which would result in a syntactically or semantically incorrect application. Based on the constraints and the generated dependency graph, DeGuard predicts the most suitable names for unknown nodes. For the prediction the relationship to known nodes is used, with a weight value. To deliver the most accurate prediction a name with maximized weight sum will be chosen. It means that for each choice the impact of the naming on the whole application is considered. The value for the weight of a relationship is computed during the learning phase, where DeGuard is trained with non-obfuscated applications found in public repositories. DeGuard can only handle layout obfuscations, which only changes the names of classes, methods and packages. Data obfuscation, control-flow and cryptographic obfuscations are currently not supported by DeGuard.

Anti-ProGuard [66] is mainly designed to identify and de-obfuscate known third party libraries. Similarly to DeGuard, it only targets the obfuscation applied by ProGuard. First, the apk-file is unpacked and the dex-files disassembled into smali files with an appropriate smali-package directory tree. Every package is analyzed, and if more than 82.5% of the classnames are shorter than 5 characters, the package is marked as obfuscated. Each obfuscated package is checked with known packages to calculate its similarity. The similarity is computed in two consecutive runs, one in a bottom-up and one in a top to bottom style. For the bottom-up similarity calculations, n-grams and Simhashes are used, whereas the top-down computation tries to map the files and methods to known ones. The final step of the de-obfuscation is to rename the identified files and methods. This

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

happens in a bottom-up fashion starting with the methods, then the classes and, last but not least, the packages. The de-obfuscation might result in broken applications, due to invalidated reflective accesses or incorrect renaming of overloaded methods, interfaces or superclasses.

3.3.3 Android Taint Analysis

Although taint analysis has already been discussed in Section 2.3.2 and mentioned in Section 3.1.1, the special nature of Android applications, like having multiple entry points require to discuss this topic separately.

Static Taint Analysis for Android

Static taint analysis in Android is not an easy task because of several challenges. Android applications have multiple entry points and multiple components, which can usually be started in any desired order. Moreover, the different components inside an application can communicate with each other through different channels, for example remote procedure calls (RPC) or Intents. This communication, just like the starting one, is not necessarily synchronous and might occur through registered callback functions.

SCanDroid [67], AndroidLeaks [68] and CHEX [69] all utilize WALA [70] to generate a call graph. SCanDroid using the generated call graph runs the flow analysis using in- and outflow filters, where the inflow filter is basically a map from inflow methods to inflow tags, and the outflow filter is a map from inflow tags to outflow tags. Parallel to the flow analysis a string analysis is started as in Android several pieces of important information, for example permissions, are stored as strings. In the end, additional metadata gathered from the manifest of the app is combined with the analysis to present the results. AndroidLeaks, on the other hand, uses a mapping between permissions and API-functions partly as a source and a sink for the taint analysis and it also marks callback functions as they are registered to see if they register to a service which can potentially leak information. CHEX divides the application into so called splits. A split is according to Lu et. al., a part of the application which can be reached from an entry point. CHEX analyzes the data-flow among the different splits, and in order to “simulate” the behavior, where different parts can be called in different orders, they run the taint analysis on every possible permutation of the splits.

FlowDroid [27], similarly to the previous ones, assumes that every component of an application can run in any order. To overcome this it uses a path insensitive analysis framework called IFDS [71]. Furthermore, FlowDroid models the life cycle of an Android app to achieve higher precision. Along the forward-taint analysis, which means propagating the taint along the path, it also does on-demand backward-alias analysis, that is upon taint propagation FlowDroid tries to find aliases on the heap along the executed path.

The biggest limitation of FlowDroid is that it cannot handle inter-component and inter-application communications. WeChecker [72], IccTA [73] and Amandroid [74] try to overcome this limitation with different methods. WeChecker uses a two-round

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

taint analysis. First, the data flow is analyzed on an intra-component basis, then the results are aggregated and used in the analysis of the whole application. IccTA, on the contrary, instruments the bytecode and changes the invocation of other components to direct calls shortcutting the event based on the nature of the framework. Although it solves the issue with calls through intents, remote procedure calls remain still untracked. Amandroid, however, provides a component-based analysis instead of a application-based one. Similarly to WeChecker it computes the data-flow for every component individually, moreover it also keeps in mind the points through which any data could enter or leave the component. In the next phase, based on the information of the point where the data can enter and leave and a component, the data flow is computed in the inter-component manner. Naturally, this approach supports the inter-application-communication tracking too, since inter-application-communication is nothing else than communication among components, but this time from two different applications.

Last but not least, DroidSafe [75] uses the so called Android Device Implementation (ADI) to achieve a very high precision. It is basically a very precise model of the Android Runtime, which includes language specific features, like inheritance or polymorphic code reuse, which in fact is heavily used by Android applications. To allow better scalability and performance DroidSafe eliminates classes which are irrelevant regarding the information flow. To provide support for asynchronous callbacks and varying order of code execution flow-insensitive information-flow analysis is used. DroidSafe supports inter-component and inter-application information-flow-tracking to some extent.

Dynamic Taint Analysis for Android

TaintDroid [31] was designed to work with the older runtime environment of Android called Dalvik. TaintDroid utilizes a 32-bit vector for storing taint tags, thus there is support for 32 different tags. It also propagates taint through JNI calls to some extent and supports inter-process communication. To prevent losing a tag after it has been written to the file system, the tag is propagated into the file as well in the form of extended attributes provided by the file system itself. During analysis the tags are stored in-memory thus resulting in a memory overhead. Figure 3.1 shows the different rules for taint propagation, where v_X is a register variable, f_X represents a class field, R is the return value, E denotes an exception, and A, B and C are byte-code constants. These rules show for example, that if a method returns a tainted variable the callee receives the tainted value, too (rule *return-op* v_A).

NDroid [76] using TaintDroid as a basis specializes in taint tracking for JNI related calls. As the authors of NDroid describe TaintDroid is limited if it comes to tracking tags in native libraries. TaintDroid can handle scenarios if a java class calls into a native library with tainted values, and as the native function returns the same class sends the return value to a source. It happens because return values, if they are related to tainted values, get tainted as well. However, in situations where for example a Java class sends tainted values to a native function and either the native function sends the value to a sink, or another Java class accesses these values by calling the native function itself, it can happen

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

Figure 3.1: TaintDroid propagation rules [31]

the other way round. In both cases TaintDroid loses the taint, as tags are not propagated so intensively in the native context. To overcome this limitation of TaintDroid, NDroid proposes the solution shown on Figure 3.2.

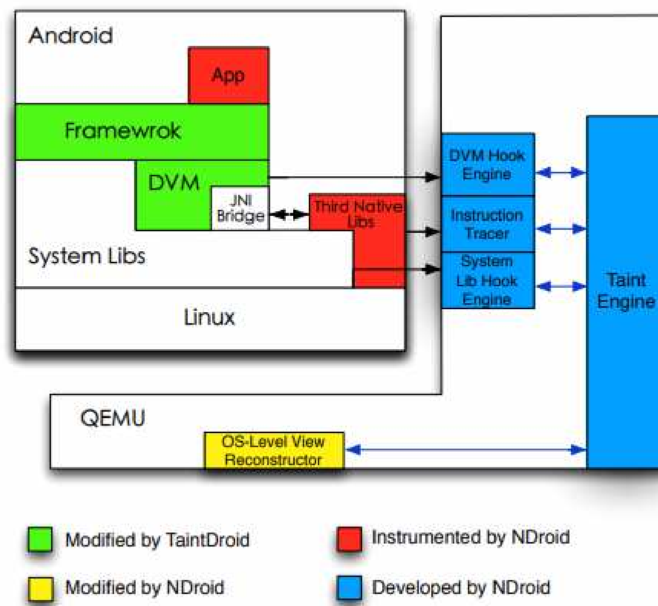


Figure 3.2: NDroid architecture [76]

It is important to notice that NDroid starts Android inside an extended version of the QEMU emulator. The taint engine keeps track of the taints in the native context with the help of shadow registers and maps for tainted memories. The most popular system library calls like *memset* get instrumented to allow taint propagation. It happens in the *System Lib Hook Engine*. JNI related functions, which take part in the context of switching between Java and native contexts, get instrumented by the *DVM Hook Engine*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

NDroid generally uses the "or" operation for propagating two different tags, which means the two tags are combined. For more information regarding propagation rules, see Figure 3.3.

Insn Format	Insn Semantics	Taint Propagation	Description
binary-op R_d, R_n , R_m	R_d = R_n op R_m	t(R_d) = t(R_n) OR t(R_m)	set R_d taint to R_n taint OR R_m taint
binary-op R_d, R_m	R_d = R_d op R_m	t(R_d) = t(R_d) OR t(R_m)	add R_m taint to R_d taint
binary-op R_d, R_m, #imm	R_d = R_m op #imm	t(R_d) = t(R_m)	set R_d taint to R_m taint
unary R_d, R_m	R_d = op R_m	t(R_d) = t(R_m)	set R_d taint to R_m taint
mov R_d, #imm	R_d = #imm	t(R_d) = TAINT_CLEAR	clear the R_d taint
mov R_d, R_m	R_d = R_m	t(R_d) = t(R_m)	set R_d taint to R_m taint
LDR* R_d, R_n, #imm	addr = Cal(R_n, #imm), R_d = M[addr]	t(R_d) = t(M[addr]) OR t(R_n)	set R_d taint to M[addr] taint OR R_n taint
LDM(POP) regList, R_n, #imm	startAddr = Cal(R_n, #imm), endAddr = Cal(R_n, #imm), {R_i, R_j} = {M[startAddr], M[endAddr]}	t({R_i, R_j}) = t(R_n) OR t({M[startAddr], M[endAddr]})	set R_i taint to M[startAddr] taint OR R_n taint, set R_{i+1} taint to M[startAddr+4] taint OR R_n taint, ..., set R_j taint to M[endAddr] taint OR R_n taint
STR* R_d, R_n, #imm	addr = Cal(R_n, #imm), M[addr] = R_d	t(M[addr]) = t(R_d)	set M[addr] taint to R_d taint
STM(PUSH) regList, R_n, #imm	startAddr = Cal(R_n, #imm), endAddr = Cal(R_n, #imm), {M[startAddr], M[endAddr]} = {R_i, R_j}	t({M[startAddr], M[endAddr]}) = t({R_i, R_j})	set M[startAddr] taint to R_i taint, set M[startAddr+4] taint to R_{i+1} taint, ..., set M[endAddr] taint to R_j taint

Figure 3.3: NDroid propagation rules [76]

TaintART [32] is one of the recently developed dynamic taint analysis systems for Android, supporting Android 5.0 and 6.0. It extends the *dex2oat* ahead-of-time compiler to instrument the already optimized native code with logic for taint propagation. It also extends the runtime environment to provide policy enforcement and to alert if tainted data would reach a sink. The tags are stored in registers in the CPU to minimize the performance overhead due to tainting. Figure 3.4 describes the different propagation rules defined by TaintART. It shows us for example that in case of binary operations the max value of the parameters' tag is taken, see rule *HBinaryOperation*.

HInstruction (Location)	Semantic	Taint Propagation Logic Description
`HParallelMove(dest, src)`	`dest` ← `src`	Set `dest` taint to `src` taint, if `src` is constant then clear `dest` taint
`HUnaryOperation(out, in)` `HBooleanNot, HNeg, HNot`	`out` ← `in`	Set `out` taint to `in` taint, unary operations ∈ {`!`, `-`, `~`}
`HBinaryOperation(out, first, second)` `HAdd, HSub, HMul, HDiv, HRem, HShl, HShr, HAnd, HOr, HXor`	`out` ← `first` ⊗ `second`	Set `out` taint to max(`first` taint, `second` taint), ⊗ ∈ {`+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`}
`HArrayGet(out, obj, index)`	`out` ← `obj[index]`	Set `out` taint to `obj` taint
`HArraySet(value, obj, index)`	`obj[index]` ← `value`	Set `obj` taint to `value` taint
`HStaticFieldGet(out, base, offset)`	`out` ← `base[offset]`	Set `out` taint to `base[offset]` field taint
`HStaticFieldSet(value, base, offset)`	`base[offset]` ← `value`	Set `base[offset]` field taint to `value` taint
`HInstanceFieldGet(out, base, offset)`	`out` ← `base[offset]`	Set `out` taint to `base[offset]` field taint
`HInstanceFieldSet(value, base, offset)`	`base[offset]` ← `value`	Set `base[offset]` field taint to `value` taint

Figure 3.4: TaintART propagation rules [31]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

CHAPTER 4

Implementation

In this chapter we describe the three main components of our system, and how they were implemented. At the end of the chapter we present the final architecture, and a summary of the systems capabilities.

4.1 Generic GATT Server

Analyzing IoT devices and their Android applications is cumbersome. Mainly because the device needs to be purchased, which might be too expensive, or it is not available for purchase at all. And secondly, Android applications get bigger and more complex, and they use obfuscation to make analysis harder. To partly overcome these obstacles and make analysis easier a generic and easily configured test device should be created. For this device a Raspberry Pi has been chosen, and the BlueZ project has been extended.

BlueZ provides an example implementation of a BLE advertiser service. However there was no other way to configure the server than hardcoding the service-, characteristic- and description-UUIDs. For this, the server has been extended with the option to parse a JSON configuration file, and loading the required UUIDs from there.

The next step after enabling an easy configuration for the advertiser, is to fill the configuration with the required UUIDs. To find UUIDs in an application both static and dynamic techniques can be used. The static method would be just to search for them in the decompiled or disassembled code, as to create a UUID one could use the `fromString` method, which requires the 128-bit representation of the UUID as a string. To find such strings one could use the regular expression in Listing 4.1.

```
[a-f0-9]{8,8}-([a-f0-9]{4,4}-){3,3}[a-f0-9]{12,12}
```

Listing 4.1: UUID regex

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Finding the required UUIDs is, however, not enough, as the context where this UUID belongs to and the fact if it is relevant at all are still missing. To connect a UUID through static analysis to a service, characteristic or a description is not really viable, which means we have to use dynamic methods again. Its details are described in Section 4.2, but for now let's assume that this information is provided as well.

This approach would have worked, if during the initial connection the user had chosen his or her device manually. The process is, however, not how it was expected. In reality the real device sends out special packages periodically to advertise it's presence. The application waits for such packages, and upon arrival it parses its content. The content of this package can include multiple information including the name of the device, the service UUIDs it has, and the custom manufacturer defined data. Apart from the custom manufacturer data the leaked UUIDs can be put on display in various forms specified by the standard to increase the chance of detectability.

Manual analysis of two Android applications (FitBit and MI Fit) has lead to the assumption, that the manufacturer data is not used during the initial device filtering. Thus faking it is not necessary. During evaluation, the absence of the manufacturer data caused no immediate problems, so it seem to confirm our hypothesis.

However, extending BlueZ to advertise the leaked UUIDs cannot overcome the limitation posed by the Bluetooth standard itself. The 4.0 Bluetooth standard allows only 31 bytes of data in the advertising package including the data length fields. UUIDs can be advertised in multiple formats, like using only their 16-bit representation of the whole 128-bit one. Using the 16-bit variant allows us to advertise 14 of the found UUIDs, as one byte of the package will be used as length indicator. But the 16-bit format can only be used if it has been issued by the Bluetooth SIG, and thus the base UUID can be omitted. Thus we only can advertise a limited amount of UUIDs.

4.2 Extending TaintDroid

TaintDroid was designed with the idea to protect the users privacy from applications which leak sensitive information to outside services. It could be GPS-coordinates or the number of the SIM card. It mainly means that the sources of the tainting were placed at points where this sensitive information was generated, and the sinks were at points like the implementation of HTTP or HTTPS sockets, where the sensitive information would leave the phone.

To use TaintDroid for analyzing the Bluetooth traffic between a phone and an external device, we had to extend the TaintDroid framework. In this section these extensions will be described.

To make our contribution distinguishable first we introduced a new type of taint tag. TaintDroid, as it was mentioned before uses 32 bit numbers as taint tags. Originally the creators of TaintDroid only used 17 of the possible 33 tags, which conveniently leaves us enough room for our extensions. The new tag was named `SSLINPUT` mostly because we

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

started extending TaintDroid by extending the implementation of the `SSLInputStream`. Although taints are only propagated in native context in a limited manner, almost the whole tainting interface is implemented as native code. This meant we had to extend both the java interface of the tainting API (`dalvik.system.Taint` in the project *framework/base*), and the native header file (`Taint.h` in the project *dalvik*) with the newly defined tag. As Android is hardwired for actually releasing the code for thousands of devices, the API-documentation had to be upgraded as well, since the new tag was exposed publicly for every other class to use.

As mentioned previously TaintDroid mainly warns the user as sensitive information on the phone leaves the device. However, what we would like to see is what comes in from the Internet and later leaves the device through the Bluetooth or gets written out to the file system. To achieve it new taint sources have been defined in HTTPS and HTTP socket implementations. During the development we noticed that a high amount of traffic was received from advertisement networks and performance monitoring services. To reduce the output of non-relevant information we implemented a filter to ignore specific sources in regard of tainting, see Listing 4.2. Currently this filter is hard-coded in the respective classes, but it should be fairly easy to read in the filter from a file.

```
private class SSLInputStream extends InputStream{
  // ...
  private String[] f = {
    "graph.facebook.com",
    "sdk.hockeyapp.net",
    "decide.mixpanel.com",
    "api.mixpanel.com"
  };

  // ...

  @Override
  public int read(byte[] buf,
      int offset,
      int byteCount) throws IOException {
    // ...
    if (hname == null) {
      // set host name to "unknown" if real one is not known
      hname = "unknown";
    }
    // Do the SSL crypto stuff like normal
    int toRet = NativeCrypto.SSL_read(sslNativePointer,
        socket.getFileDescriptor$(),
        OpenSSLSocketImpl.this,
        buf,
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
        offset,
        byteCount,
        getSoTimeout());

    ArrayList<String> filter = new ArrayList<String>(
        Arrays.asList(f)
        );
    if (!filter.contains(hname)) {
      // do tainting
    }

    return toRet;
  }

  // ...
}
```

Listing 4.2: Don't taint traffic for specific domains

Extending the original Android code for implementing a sink at HTTP and HTTPS socket implementations, like the original authors of TaintDroid did, is easy. The only thing to do is, to check if the byte array, which is to be written into the socket, has any taints, and if so to generate a warning to notify the user. However, implementing a source at these places turned out to be more complicated as it had been expected. The reason for this was, that although the tainting worked for incoming traffic, as soon as the `Content-Encoding` header was set for example to *gzip* the taint was lost at some point. The cause of it was that the inflater allocated a new byte array for the inflated data, which was filled in the native context, as for this, the *zlib* library was used. As it has already been stated multiple times taint tags are not propagated by default in native context except for some special cases, and this resulted in the loss of the tag. To correct it the responsible java class was extended to copy the tag of the input array to the output if it was needed.

As mentioned in the previous section searching for UUIDs is essential if one would try to simulate the presence of a BLE device. Although the previously mentioned `fromString` method requires a well defined and easily searchable string, there is another way to create UUIDs, which is not easily searchable. This would require to pass two long integers through the constructor of the UUID class representing the most and least significant bits of the UUID. To catch these instances as well, and also other edge cases like using a mask and another string generating the UUID at runtime, we extended both the constructor and the `fromString` methods to leak the generated UUIDs, see Listing 4.3.

```
public final class UUID implements Serializable,
        Comparable<UUID> {
  public UUID(long mostSigBits, long leastSigBits) {
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
        this.mostSigBits = mostSigBits;
        this.leastSigBits = leastSigBits;
        init();
        Taint.log("UUID-LEAK: " + this.toString());
    }

    // ...

    public static UUID fromString(String uuid) {

        // ...

        Taint.log("UUID-LEAK: " + uuid);
        return new UUID(msb, lsb);
    }

}
```

Listing 4.3: Leaking UUIDs as soon as they are created

The last piece of the puzzle is to connect the UUIDs to the actual resources of relevance, namely the services, characteristics and descriptions. Fortunately, the Android interface requires to pass the UUID of that resource to access a service, a characteristic or a descriptor. It means to extend the classes representing the aforementioned resources to leak the missing information we can connect the UUIDs to the type of the resource to use it later.

Last but not least, the different types of Bluetooth classes, which are able to send out information, have been extended to be taint sinks. That is if tainted information leaves the phone through either Bluetooth Classic or Bluetooth Low Energy, the user will be notified. Additionally to the notification, if the data was tainted with the `SSLINPUT` tag, it will be written out to a predefined path on the file system. We think this is necessary as the update files are also transmitted over Bluetooth to the peripheral device, which could be a source of more information during the analysis. On the top of all the call trace is also dumped through the logs to help the analyst to identify where the writing request came from within the application.

4.3 Aggregating Results

Although the previously mentioned pieces of information are presented to the user through the Android logs, so are several other unrelated entires as well, which should be filtered out. To aggregate and display the information in an organized manner a python framework has been developed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

The framework uses the Android log reader facility called `logcat` to read the logs. As soon as an information, like a UUID has been leaked through the logging mechanism of Android the framework picks it up and stores it. If possible, the information is categorized to make the life during analysis easier. It means as soon as a UUID turns out to be used as a service identifier, the framework stores this information as well. Additionally, the number of occurrences of the UUIDs is also stored, as this way potentially false positive UUIDs can be identified and excluded.

Besides the UUIDs specially crafted stacktraces are collected as well, since they can pinpoint where the Bluetooth related code can be found in an application. During visualization of the stacktraces if a mapping file from old names to new names is provided in a special format, the relevant parts of the stacktrace gets rewritten. This is relevant if the application gets de-obfuscated, but to counteract potential information losses, like dynamic reflection due to renaming, the original application is used for the dynamic analysis and the de-obfuscated application is only analyzed statically.

To obtain the previously mentioned mapping file the framework provides a binding to the external tool called `apk-deguard` which perform de-obfuscation by applying statistical methods. As a result apk-deguard provides a rewritten APK and decompiled Java sources as well besides the mapping, however, the previous two are at the moment not used directly by the framework..

4.4 Final Architecture

Figure 4.1 depicts the final architecture of the implemented framework. As an input we use the downloaded APK of the application we would like to analyze. It gets installed on the test phone, which runs the modified version of the TaintDroid taint-analysis framework. Optionally the APK can be loaded into the collector as well. When it happens the APK is sent to the DeGuard framework to undo the obfuscation applied by ProGuard, which is the default obfuscator written for Android.

The phone acts as a middle-man and relays the communication between the service backend on the Internet and the Bluetooth fitness device, as it was intended by the developers of the application. Meanwhile, the collector collects specially tagged log entries, generated by TaintDroid. As soon as the analysis is considered done, the collected information gets sorted and is presented to the user in a structured format.

The framework offers limited options to simulate the presence of an actual Bluetooth device. To use this capability a preliminary analysis is required through which it can be determined what kind of Bluetooth Low Energy services and characteristics are expected to be present. If this analysis is done the system can be configured easily to include the given services with the given characteristics. This subsystem is not only able to show the data which was sent to it by the phone, but can send back user defined data as well, as if it were a standard Bluetooth device.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

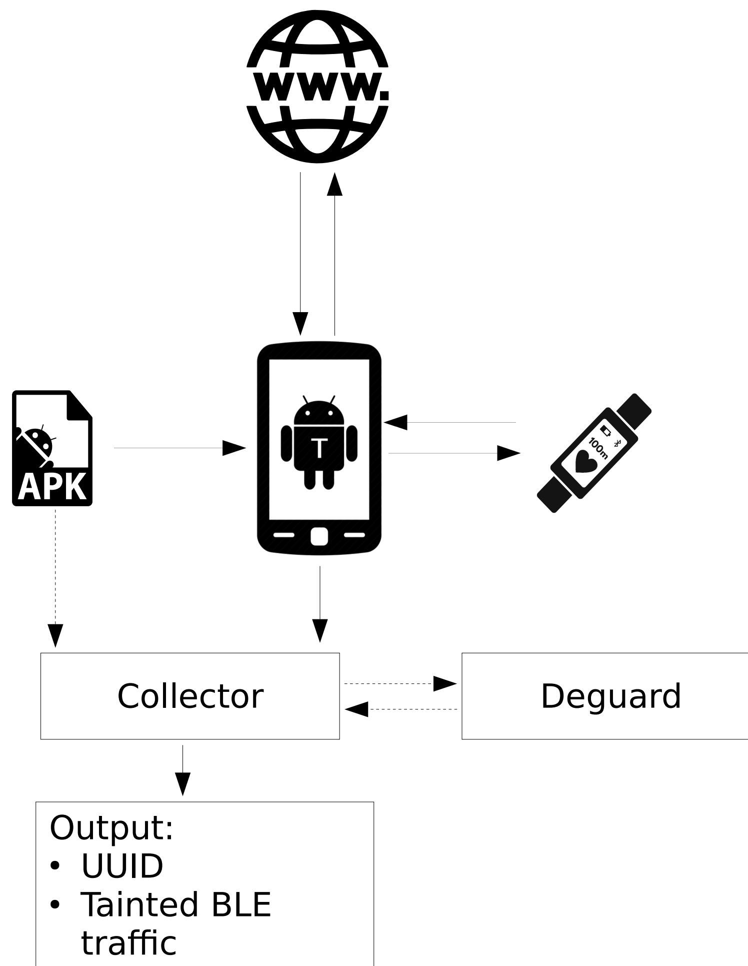


Figure 4.1: Architecture of the implemented system

4.4.1 Degree of Automation

The current system was designed to conduct human assisted analysis, thus it still heavily relies on human work. Currently the application is capable of collecting and identifying known Bluetooth UUIDs by mapping them to the database of the Bluetooth SIG assigned numbers. Besides the UUIDs, if tainted data is written out through the Bluetooth interface, a warning is generated to help identifying the relevant classes inside the application which are responsible for handling the Bluetooth traffic. These warnings are collected and presented by the framework. To enable further analysis of such a tainted data, before sending it to the device it gets written out to the file system of the phone

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

and through the framework it can be downloaded easily.

If DeGuard had been used, these trace would have no value, as the classes are partially or completely renamed. That is why we use the mapping file provided by DeGuard to rewrite the collected traces to the new names.

On the Raspberry Pi, the extended version of the GATT-Server of BlueZ can handle JSON configuration files, thus setting up the required services and characteristics is relatively easy. Some applications require the device to advertise its presence either by broadcasting specific information like a service UUID or by a specially chosen device name. Either way this is left to be configured by the user, using the management tools provided by the BlueZ project.

Furthermore extracting and analyzing the protocol used by the device and the application to communicate with each other is not automated yet, and is required to be done by the user conducting the analysis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 5

Evaluation

In the first part of this chapter we are going to describe the applications we used to test our framework. In the second part we present the results of our tests with the chosen applications.

5.1 Sample Set

Initially, the implemented framework was tested with a self written Android application. After starting the application it searched for the Bluetooth MAC of the test device, which was a Raspberry Pi with some readable and writable BLE characteristics. As soon as the Raspberry Pi had been found, the app connected to it and downloaded the front page of "Google". This way we acquired some data which got tainted with the newly implemented tag. This data was then sent to one of the BLE characteristics, which - as expected - generated a warning in the logging system that the tainted data had left the device through a Bluetooth sink.

This application was really helpful during the development, however, the logic and the size does not represent real applications. In the following few paragraphs real life applications will be described shortly.

Through the *Fitbit* application the user can set up and remove devices connected to the account. It also manages the synchronization between the connected devices and the account stored on the servers provided by the Fitbit company. The application also fetches the logged data from the server to present it for the user. Besides the data generated by the connected device the user is able to log additional information like water or food consumption and weight. The authors had access to an older Fitbit device, the Fitbit One, which was partially used for testing.

Very similarly to Fitbit, *MI Fit*, implemented by the Chinese company Xiaomi, manages the connections between a phone and devices like the *MI Band*. MI Fit is, however, not so

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

feature packed as Fitbit. The user can only log his/her weight as additional information. Besides that and by setting a daily goal for step count a workout session can be tracked, where data like GPS, distance, heart rate and step counting are recorded with a higher precision of time. During the evaluation the authors had access to an older version (2) MI Band.

Mio GO is a plain workout tracker with the main focus on heart rate monitoring. The Mio devices are advertised as strapless ECG accurate heart rate monitors. Although the app is relatively minimalist in features and design the user is not required to have a Mio GO account. Additionally, unlike the previous apps with Mio GO exporting the data into CSV is supported natively.

Polar produces different fitness trackers ranging from pure heart-rate monitors to sport watches. To download the data from these devices the user is required to use either *Polar Flow* or *Polar Beat*. Polar Beat is designed to track individual training sessions and pull "*live*" data from the fitness device, whereas Polar Flow serves as an overview and data accumulator for day-to-day usage beyond a training session. It is worth to mention, that Polar Flow is only compatible with selected Polar devices, whereas Polar Beat supports not only Polar HR monitors but works with third party sensors as well. However, in case of third party sensors, Polar Beat only offers real-time heart rate monitoring and features like burnt calories or fitness tests are restricted.

Misfit is an everyday activity and sleep tracking application written to be used with wearables manufactured by the same-named US based company. Besides tracking basic activities, like steps-per-day, the users can specifically tag movements where they are doing some sort of sport in the form of a workout like swimming, cycling or even yoga. To encourage more movements users can set daily goals, and the application sends notifications every now and then to remind them, that it is time for the movement. Misfit also offers social experience in the form of competing against friends and other users and also by sharing results with others.

5.2 Analysis Results

For each previously described application the following evaluation steps have been applied. First without any device emulation or real device nearby the app has been started. Then an initial binding step is initiated as if someone connected the application to a device. Through it the application is going to search for a device eventually resulting in a timeout as no device is nearby. Although the initialization of the connection fails the analysis framework records valuable information.

At this point we can only evaluate the false discovery rate of the leaked UUIDs, as there was nothing to connect to, hence there can be no call trace leaks, as that code never becomes executed. To be able to count the false positives, without the need of manual reverse engineering at this point, we execute the analysis three times in total. Between each execution the application in test is going to be terminated and the logs on the phone

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

are going to be purged to eliminate the contamination of results and to prevent the reuse of UUIDs generated randomly. This way we have a way to compare which UUIDs appear even after restarting the application, which is an indicator for the UUID being created specifically and not randomly.

Besides determining the false discovery rate of the UUIDs we can use the most likely true positive UUIDs for the next step, which are trying to emulate a device.

Last but not least, the effectiveness of the information retrieval can be evaluated with the help of a real device as well.

5.2.1 MI Fit

At the first test, trying to make a connection without a device, the framework was able to find 33, 39 and 37 unique UUIDs for the three analysis executions respectively, which were used by the application. None of the found UUIDs were categorized as service- characteristic- or descriptor UUIDs, and no call trace was recorded, as it had been expected.

Eleven UUIDs were found in all the three sets of the recorded UUIDs, they are listed in Table 5.1. The table also shows what that UUID can be used for according to the Bluetooth specification or which SIG member it belongs to. Correlating the results of the three analysis results we were able to prune around 70% of the irrelevant UUIDs.

Filtering with the help of open information provided by the Bluetooth SIG we can identify and categorize some of the eleven UUIDs found in every execution. The UUIDs with *0000fee0* and *0000fee1* belong to the company *Anhui Huami Information Technology Co., Ltd.* which is the manufacturer of the Xiaomi devices. Manual reversing, online research and inspecting the actual device confirmed that these are two of the three custom services on the MI Band 2. The UUID with *0000fee7* belongs to another Chinese company called *"Tencent"*. Manual reverse engineering revealed that this service is used for showing messages sent through the Chinese messaging and social media service *"WeChat"*. The last two UUIDs beginning with *00002901* and *00002902* belong to two standardized descriptors namely *"Characteristic User Descriptor"* and *"Client Characteristic Configuration"*. Former is used, as the name suggests, to provide a human readable description the way the given characteristic it belongs to does, and latter holds a client specific configuration to enable or disable notifications and indications. The remaining common UUIDs neither could be categorized by the framework nor have been found through manual online research, except the UUID with *f0000000* which is the base UUID for devices manufactured by the company *"Texas Instruments"*. However, manual reverse engineering revealed no usage for these UUIDs in any way, and the real device did not include them either.

It means after the initial cleanup, i.e. correlation of the three executions, the framework had a false discovery rate of 55%.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
00000000-0000-3512-2118-0009af100700	-
00000000-1212-efde-1523-785feabcd123	-
000009f0-0000-1000-8000-00805f9b34fb	-
00000af0-0000-1000-8000-00805f9b34fb	-
00002901-0000-1000-8000-00805f9b34fb	Characteristic User Description
00002902-0000-1000-8000-00805f9b34fb	Client Characteristic Config.
0000fee0-0000-1000-8000-00805f9b34fb	Anhui Huami IT Co., Ltd.
0000fee1-0000-1000-8000-00805f9b34fb	Anhui Huami IT Co., Ltd.
0000fee7-0000-1000-8000-00805f9b34fb	Tencent Holdings Limited.
cbbfe0e1-f7f3-4206-84e0-84cbb3d09dfc	-
f0000000-0451-4000-b000-000000000000	Texas Instruments*

Table 5.1: Similar UUIDs throughout three analysis executions and their meaning (entries with * were categorized manually)

The next step was to try to simulate the presence of a real device. At this point we had three service UUIDs (two registered to Huami and one to Tencent) and we could advertise those with the Raspberry Pi method which was previously described in Section 4.1.

Our log analysis framework picked up a clear pattern for trying to access a non-existent characteristic, see Listing 5.1. As we could see the app grabbed the *0000fee1* service and tried to set up a notification listener for the characteristic *00000009*. After re-configuring the Raspberry Pi by following the same principle looking for patterns provided by our framework, we were able to mimic the presence of a Mi Band 2.

```
Found SERVICE_UUID: 0000fee1-0000-1000-8000-00805f9b34fb
Found CHARACTERISTIC_UUID: 00000009-0000-3512-2118-0009af100700
Found DESCRIPTOR_UUID: 00002902-0000-1000-8000-00805f9b34fb
```

Listing 5.1: Pattern to access a specific characteristic

Figure 5.1 shows that the application actually sends some data to the handle *0x00e9* which in fact corresponds with the characteristic with the UUID *00000009*.

```
Connect from BC:F5:AC:60:B4:57
Running GATT server
[GATT server]# Desc config Enabled for handle 0x00ea: true
[GATT server]# Received 18 bytes on handle 0x00e9
[GATT server]# Data received (as string):
[GATT server]# Data received (as hex): 0x01 0x00 0x06 0x70 0x59 0x66 0x23 0xa0 0x90 0xc2 0x45 0x47 0x8e 0x10 0x72 0xda 0xe6 0x16
[GATT server]# Desc config Enabled for handle 0x00ea: false
[GATT server]# Device disconnected: Connection reset by peer
```

Figure 5.1: First package after connection

With the help of the applications internal logs and an open source project[1] we were able

[1] https://github.com/creotiv/MiBand2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to send back a valid response. The first package which we saw was sending an encryption key from the app to the device. The device had to send back the acknowledgment of receiving the encryption key. The application then asked for 16 random bytes, which were then encrypted by the application and the encrypted bytes were sent back for verification. After receiving the last acknowledgment the application read out the device information like the serial number, the hardware and software revisions, from the standardized BLE characteristics. For these stats we just returned zeros.

At this point the application reported that the pairing had been successful and showed the device options on the GUI. However, the connection was canceled. If we had moved back on the GUI the paired device would have disappears. According to the logs of the application, the device manager was missing on our device. In the future we are going to look into it, why the pairing failed, even though the GUI showed that it was successful.

Last but not least, a real MI Band 2 was used to test the performance of the framework. The MI Band was set back to factory defaults, and during the test the initial pairing to a registered account was started.

Immediately after pairing the real MI Band 2 to our test account the applications notified us, that there was a firmware update available. Although we could see in both our framework and the Android logs that two characteristics are being read and written rapidly, the framework was not able to dump any data written out through the Bluetooth interface. The reason for this was, that we suspected, that the firmware images would be downloaded from the Internet, and thus would be tagged with our taint tag, and we only would dump tainted data to minimize the false positive dumps. However, MI Fit packed the firmware update files into the APK, thus avoiding to be tainted, which in fact prevented the framework to dump the images.

5.2.2 Fitbit

Unfortunately, Fitbit dropped the support for devices running pre-Android 5.0 operating systems in the third quarter of 2018, thus we were required to use an older version (version 2.19) of the application, which was downloaded from a third party website specialized in providing different versions of different applications. Although there would have been newer compatible versions, the most up to date one resulted in several crashes. Because of this, starting with the oldest version of the application the first working version was been chosen.

Before starting the previously defined sequence of tests to evaluate the performance of the framework on Fitbit, the Fitbit application was started without any assigned user. It means the application could only reach the login screen. To this point the framework had already picked up 46 unique leaked UUIDs from which 5 were identified as an existing Bluetooth SIG issued UUID.

During logging into the newly created account more than a 1000 UUIDs were picked up, however, these were with high certainty randomly generated and used for other purposes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

than Bluetooth, as UUIDs are used not only for graphical user interface (GUI) related actions, but also for parcels, which are the effective ways of transferring data among applications or modules of an application like a service. It means we most likely will not have to deal with this daunting amount of UUIDs.

The three executions without any nearby device and without any device trying to emulate the presence of a real device resulted in 311, 344 and 346 UUIDs respectively. Between the three independent executions 50 common UUIDs were identified from which the framework automatically identified 24 known UUIDs. Table 5.2 shows the identified UUIDs defined by the Bluetooth SIG. Interestingly enough, a UUID (last row in the table) was picked up as well which actually belonged to MI Fit devices. This happened because the actual MI Band 2 device was in the scanning range of the test phone, and it was advertising its presence. To confirm this hypothesis, a fourth execution was performed with the MI Band removed from the Bluetooth range of the phone, and as suspected it was not present in the logs either.

Table 5.3 shows the UUIDs which were common for the three executions, although were not identified by the framework. With manual online research most of the UUIDs were identified. The manually identified list contained a relatively high number of Apple related UUIDs. This can be explained in the same way why the MI Band related UUID was found in the logs. Some Apple devices were in the scanning range of the test phone. Unfortunately, these devices were not removable from the range because they did not belong to the author and the author had no way to shield the phone used for testing from picking up “rouge” devices.

Through the initial cleanup step, by correlating the three executions we were able to eliminate around 85% of the not relevant and probably randomly generated UUIDs. From the remaining 50 UUIDs the framework was able to identify, i.e. connect the UUID to a human understandable meaning, almost 50%. After manual identification of the UUIDs the identification rate rose to 80%. Considering the application relevant UUIDs (including the UUIDs defined by the Bluetooth SIG, as they could be found in the APK, with exception of the manufacturer UUIDs) as true positives and everything else as false positives, the false discovery rate of the framework was 32%.

Although the framework unfortunately could not identify these UUIDs alone, we were going to use them nonetheless for the second stage of our evaluation tests, namely trying to emulate the presence of a real Fitbit One step and floor counter device. It should be mentioned that the manually identified UUIDs were added to the frameworks database, so this way they will be identified during future analysis runs.

The previous step of the analysis revealed 3 relevant services, namely *adabf00* which was believed to be the control service of Fitbit devices, through which the phone can send commands and receive their result, *558dfa00* which served the collected fitness data, and last but not least, *16bcfd00* which was a service on the phone and to which the real device connected back. All of these three services would be advertised and created on the Raspberry Pi along with their corresponding characteristics.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
00000000-0000-1000-8000-00805f9b34fb	BASE_UUID
0000000f-0000-1000-8000-00805f9b34fb	BNEP
00001105-0000-1000-8000-00805f9b34fb	OBEXObjectPush
00001108-0000-1000-8000-00805f9b34fb	Headset
0000110a-0000-1000-8000-00805f9b34fb	AudioSource
0000110b-0000-1000-8000-00805f9b34fb	AudioSink
0000110c-0000-1000-8000-00805f9b34fb	A/V_RemoteControlTarget
0000110d-0000-1000-8000-00805f9b34fb	AdvancedAudioDistribution
0000110e-0000-1000-8000-00805f9b34fb	A/V_RemoteControl
00001112-0000-1000-8000-00805f9b34fb	Headset - Audio Gateway (AG)
00001115-0000-1000-8000-00805f9b34fb	PANU
00001116-0000-1000-8000-00805f9b34fb	NAP
0000111e-0000-1000-8000-00805f9b34fb	Handsfree
0000111f-0000-1000-8000-00805f9b34fb	HandsfreeAudioGateway
00001124-0000-1000-8000-00805f9b34fb	HumanInterfaceDeviceService
0000112f-0000-1000-8000-00805f9b34fb	Phonebook Access - PSE
00001132-0000-1000-8000-00805f9b34fb	Message Access Server
00001133-0000-1000-8000-00805f9b34fb	Message Notification Server
00001134-0000-1000-8000-00805f9b34fb	Message Access Profile
00001800-0000-1000-8000-00805f9b34fb	Generic Access
00001812-0000-1000-8000-00805f9b34fb	Human Interface Device
00002902-0000-1000-8000-00805f9b34fb	Client Characteristic Config.
00002a00-0000-1000-8000-00805f9b34fb	Device Name
0000fee0-0000-1000-8000-00805f9b34fb	Anhui Huami IT Co., Ltd.

Table 5.2: Similar UUIDs throughout three analysis executions and their meaning for the Fitbit application

After creating the previously mentioned services and characteristics, first the control service was advertised. The reason why not every service was advertised is two-fold. As the UUIDs were not issued by the Bluetooth SIG, the device most unlikely advertised only with a 16-bit UUID, as it might have collided with a UUID which was issued by the Bluetooth SIG and this way might have violate the uniqueness of the ID. To overcome it one can advertise the full 128-bit UUID, however, this takes up 18 bytes (16 bytes UUID, 1 byte length field and 1 byte data type field) from the available 31 bytes according to the Bluetooth standard. This means only one service can be advertised at the same time.

After setting up the advertisement for the designated service the connection procedure was initiated. Observing the logs of the application it actually found the Raspberry Pi and identified it as a valid Fitbit device, however, the application tried to determine unsuccessfully the type of the device. Manual reverse engineering of the relevant parts

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
066bd727-1864-4bd4-8f95-bc5089b52f5d	-
16bcfd00-253f-c348-e831-0db3e334d580	Fitbit Service
16bcfd01-253f-c348-e831-0db3e334d580	Fitbit Characteristic
16bcfd02-253f-c348-e831-0db3e334d580	Fitbit Characteristic
16bcfd03-253f-c348-e831-0db3e334d580	Fitbit Characteristic
16bcfd04-253f-c348-e831-0db3e334d580	Fitbit Characteristic
1880d30a-1fbe-44c2-903d-0d061f5a9c12	-
22eac6e9-24d6-4bb5-be44-b36ace7c7bfb	Apple Data Source
4cedb2a9-8aea-40fb-851a-6ed0783c7a28	-
558dfa00-4fa8-4105-9f02-4eaa93e62980	Fitbit Data Service
558dfa01-4fa8-4105-9f02-4eaa93e62980	Fitbit Data Characteristic
69d1d8f3-45e1-49a8-9821-9bbdfdaad9d9	Apple Control Point
6c43da2d-e71d-4db3-83e2-57657a129bd9	-
6dff7b09-b857-4d36-b5c0-b1b27b42edd6	-
7905f431-b5ce-4e99-a40f-4b1e122d00d0	Apple Notif. Center Service
88492498-ccd7-42d5-88b2-59dd4697dc83	-
9fbf120d-6301-42d9-8c58-25e699a21dbd	Apple Notif. Src. Characteristic
adabfb00-6e7d-4601-bda2-bffaa68956ba	Fitbit Ctrl. Service
adabfb01-6e7d-4601-bda2-bffaa68956ba	Fitbit Ctrl. Read Characteristic
adabfb02-6e7d-4601-bda2-bffaa68956ba	Fitbit Ctrl. Write Characteristic
adabfb04-6e7d-4601-bda2-bffaa68956ba	Fitbit Ctrl. Characteristic
b4766dea-60d5-4b53-a304-7710ce416689	-
c09ee05a-e1fb-434a-b7ba-7da41441917c	-
c98e7d81-7213-4dcb-afa0-fde52962f2ba	-
ffff0000-ffff-ffff-ffff-ffffffffffff	UUID mask
fffffde0-0000-1000-8000-00805f9b34fb	-

Table 5.3: UUIDs which were found in three executions of the Fitbit application although they were categorized manually

of the application revealed that at a specific position in the advertising data the type identifier of the device was to be expected. One way to advertise such an information is using the *Device information* service. After finding the device ID mappings in the application and starting a new advertisement this time with the Device information service activated with the relevant value, the application recognizes the Raspberry Pi as an actual Fitbit One and tries to connect to it.

The connection is successful and can be verified on the Raspberry Pi as well. The application even shows a message "*We have found your device*", although as the application advances to the next phase which is to display the connection code, to verify that the correct device has been found the connection is suddenly terminated. Regrettably, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

logs of the phone do not provide any indication why the connection broke, but in any case it is initiated by the phone, like Figure 5.2 shows.

```
Started listening on ATT channel. Waiting for connections
Connect from BC:F5:AC:60:B4:57
Running GATT server
[GATT server]# Device disconnected: Connection reset by peer


Shutting down...
```

Figure 5.2: GATT-Server disconnection during Fitbit simulation

The reason for the disconnection was that the *Fitbit Control Read Characteristic* had no Client Configuration Descriptor and this way the app could not subscribe for notifications. After the reconfiguration we received 13 bytes from the phone on the *Fitbit Control Write Characteristic*, see Figure 5.3. However, after around 10 seconds the connection was interrupted again because of a timeout as we could see on the logs of the phone. If we had send back the same 13 bytes through the Control Read Characteristic, the app would have printed out an error message through the logs, that the received bytes could not be decoded. Searching for the printed string we could locate the source code of the response parsing utility, through which we can create a valid package that would have been accepted by the application.

```
Running GATT server
[GATT server]# Desc config Enabled for handle 0x000a: true
[GATT server]# Received 13 bytes on handle 0x000c
[GATT server]# Data received (as string): �


[GATT server]# Data received (as hex): 0xc0 0x0a 0x0a 0x00 0x08 0x00 0x10 0x00 0x00 0x00 0xc8 0x00 0x01
```

Figure 5.3: GATT-Server receives the first package from Fitbit

With the help of the decompiled application and the open source project *galileo*[2] we were able to decode the package. First, the application sent a request to initialize the Fitbit AirLink. As a response we sent back an AirLink established package which included the tracker ID, which was the MAC-address of the device in little endian format and the set of parameters. Next, the app sent a *display code* package, which would trigger off showing the pairing PIN on a real device, but we just sent back an acknowledged response. During the next step the app requested a minidump from our device which was - depending on the device - either AES or XTEA encrypted. We tried to replay a valid package found on the Internet which was accepted by the application, but this information is sent to the Fitbit backend server for validation, which as we would suspect failed. As we did not know the content of a valid minidump we could not possibly fake a valid Fitbit device.

[2] https://bitbucket.org/benallard/galileo

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Last but not least, we paired a real Fitbit One with our test account, and started the firmware update process. In the logs of our test phone we could confirm, that data is downloaded from the Fitbit backend server, and thus got tainted with our custom tag. A few seconds later a progress bar appeared on the screen of our test Fitbit One, see Figure 5.4, and soon we saw in the framework, that data sent out through the BLE interface was dumped 20 bytes a time, as this was the maximum payload size for Bluetooth Low Energy packages prior to Bluetooth 5.0.

Figure 5.4: Firmware update progress on a Fitbit One

After writing and dumping around 3000 bytes, the firmware update process got interrupted due to a segmentation fault in the `getTaintFile`. We are going to look into this issue in the future. Nevertheless we had the first few thousand bytes of the new firmware, the remaining part was, however, not accessible for us as the firmware image was handled in memory and was never written out to the file system.

Figure 5.5 showed the entropy graph of the first thousand of bytes of the dumped firmware image. The high entropy indicates that the data is encrypted, which can be partially confirmed, since the logs on the phone show that the data is transmitted as microdumps, which are, as stated before, encrypted with AES or XTEA. The encryption key is not stored on the phone as the traffic between the backend and the device is end-to-end encrypted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

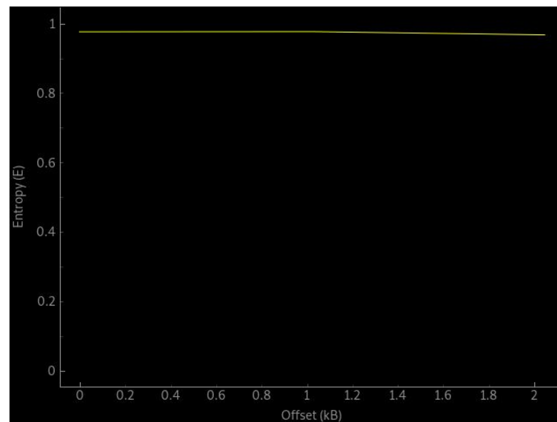


Figure 5.5: Entropy of the Fitbit firmware image

5.2.3 Mio Go

For evaluation we used the most recent version (2.7.4.4) of Mio Go as of writing this thesis.

At the startup of the application the user is greeted with a registration form, when some UUIDs were picked up by the framework, but at this point of time neither consistency could be found, nor did the framework identify any standardized UUIDs, which suggested that these UUIDs were most likely not relevant to Bluetooth related analysis. Right after completing the registration and filling out the basic user information, like name, height and weight the application started a Bluetooth scan to find nearby devices. At this point the framework was able to identify 16 of the 43 unique UUIDs and, although some of the UUIDs were not identified automatically, they had already shown a pattern where only the 16-bit part of the UUID differed and both the prefix and the postfix of the UUIDs were the same, see Table 5.4. Even without the three independent executions, one can argue that these UUIDs with most certainty were not generated randomly.

Like earlier we also executed the application three times and correlated the results. To make sure that the UUIDs would not get reused like previously between the executions both the framework and the application were restarted, and additionally, the application was force stopped to make sure to kill even the background services which could potentially preserve UUIDs and would spoil the results.

The test executions with searching for nearby devices yielded from the beginning of the execution until the end of the scan procedure to 127 UUIDs for the first execution and 129 UUIDs for the second and the third one. Between the three executions 29 common UUIDs were found which are presented in Table 5.5. As we had suspected, the UUIDs with the postfix *381c08ec57ee* were not randomly chosen, but most likely belonged to real Mio devices, and were created to be used as search filters.

From the 29 common UUIDs the framework was able to identify 16 UUIDs. Conveniently,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
6c721530-5bf1-4f64-9170-381c08ec57ee	-
6c721531-5bf1-4f64-9170-381c08ec57ee	-
6c721532-5bf1-4f64-9170-381c08ec57ee	-
6c721550-5bf1-4f64-9170-381c08ec57ee	-
6c721551-5bf1-4f64-9170-381c08ec57ee	-
6c721552-5bf1-4f64-9170-381c08ec57ee	-
6c721553-5bf1-4f64-9170-381c08ec57ee	-
6c721838-5bf1-4f64-9170-381c08ec57ee	-
6c722a80-5bf1-4f64-9170-381c08ec57ee	-
6c722a82-5bf1-4f64-9170-381c08ec57ee	-
6c722a83-5bf1-4f64-9170-381c08ec57ee	-
6c722a84-5bf1-4f64-9170-381c08ec57ee	-

Table 5.4: Unknown UUIDs with high similarity found during the execution of Mio GO

the Mio GO application was not obfuscated and so the remaining UUIDs were easily found and identified in the application. These UUIDs are marked with an asterisk in Table 5.5. To yield a better performance during future analyses the identified UUIDs were added to the framework's identification database. Additionally to the UUIDs in the previously mentioned table, three more service UUIDs were found which were not detected, as they were stored as strings instead of as UUID objects, thus had never been leaked. Interestingly enough, these IDs were seemingly never used in the application. The additional 3 UUIDs are shown in Table 5.6.

Considering these results, 76% of the leaked UUIDs were not relevant for the analysis, as they were generated mostly randomly. Although only around 50% of the correlated results were automatically identifiable, the rest were easy to be identified manually with plain text search, as we knew what we were looking for and luckily the application was not obfuscated, which is really rare for professional applications. After the manual identification it turns out that every correlated UUID is in use.

The next step was to set up the Raspberry Pi with the newly found service and characteristic UUIDs. However, starting a GATT server with just the correct UUIDs was not enough, as the application could not identify the Raspberry Pi as a Mio device. But after analyzing the decompiled application it turns out, that the application only filters by the name of the Bluetooth device, which is easy to set and fake. After setting the Raspberry Pis Bluetooth name to "ALPHA2_OTA" the app identifies the Raspberry Pi as a Mio device, as Figure 5.6 shows.

After pairing the phone and the faked Mio Alpha2 the application greets us with a new activity, namely that there is an update for our device, see Figure 5.7. This is actually due to the fact, that although we indeed set up a GATT server, we only returned empty strings, and apparently the application only compared if the returned value was the same

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
00001533-1212-efde-1523-785feabcd123	DFU Status report*
0000180a-0000-1000-8000-00805f9b34fb	Device Information
0000180d-0000-1000-8000-00805f9b34fb	Heart Rate
0000180f-0000-1000-8000-00805f9b34fb	Battery Service
00001816-0000-1000-8000-00805f9b34fb	Cycling Speed and Cadence
00002902-0000-1000-8000-00805f9b34fb	Client Characteristic Config.
00002a19-0000-1000-8000-00805f9b34fb	Battery Level
00002a23-0000-1000-8000-00805f9b34fb	System ID
00002a24-0000-1000-8000-00805f9b34fb	Model Number String
00002a25-0000-1000-8000-00805f9b34fb	Serial Number String
00002a26-0000-1000-8000-00805f9b34fb	Firmware Revision String
00002a27-0000-1000-8000-00805f9b34fb	Hardware Revision String
00002a28-0000-1000-8000-00805f9b34fb	Software Revision String
00002a29-0000-1000-8000-00805f9b34fb	Manufacturer Name String
00002a37-0000-1000-8000-00805f9b34fb	Heart Rate Measurement
00002a5b-0000-1000-8000-00805f9b34fb	CSC Measurement
0000000c-0000-1000-8000-00805f9b34fb	HTTP
6c721530-5bf1-4f64-9170-381c08ec57ee	DFU Service*
6c721531-5bf1-4f64-9170-381c08ec57ee	DFU Control point*
6c721532-5bf1-4f64-9170-381c08ec57ee	DFU Packet*
6c721550-5bf1-4f64-9170-381c08ec57ee	Alpha2 DFU Service*
6c721551-5bf1-4f64-9170-381c08ec57ee	Alpha2 DFU ctrl point*
6c721552-5bf1-4f64-9170-381c08ec57ee	Alpha2 DFU send packet*
6c721553-5bf1-4f64-9170-381c08ec57ee	Alpha2 DFU packet*
6c721838-5bf1-4f64-9170-381c08ec57ee	Mio Sports Service*
6c722a80-5bf1-4f64-9170-381c08ec57ee	Mio Sport Message*
6c722a82-5bf1-4f64-9170-381c08ec57ee	Mio Sport Message response*
6c722a83-5bf1-4f64-9170-381c08ec57ee	Mio Sensor*
6c722a84-5bf1-4f64-9170-381c08ec57ee	Mio record*

Table 5.5: Common UUIDs along three independent devices search in Mio GO (rows with * were identified manually)

as the Mio's backend server served.

Our extension to TaintDroid, as had been expected, tainted the downloaded ZIP archive containing the firmware image to our faked device. As the file was downloaded and written to the file system, TaintDroid propagated the taint information to the written out archive. When we pulled this archive from our test phone and tried to unzip it, we experienced, that it was password protected. Searching for the password in the application is, of course, a valid method and in case of a not obfuscated application like

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
381C08EC-57EE-0611-0601-02362E315055	Mio Fuse DFU Service
381C08EC-57EE-0611-0601-02312E315055	Mio Link DFU Service
381C08EC-57EE-0611-0601-02332E315055	Mio Velo DFU Service

Table 5.6: Additional UUIDs which are never used in Mio GO

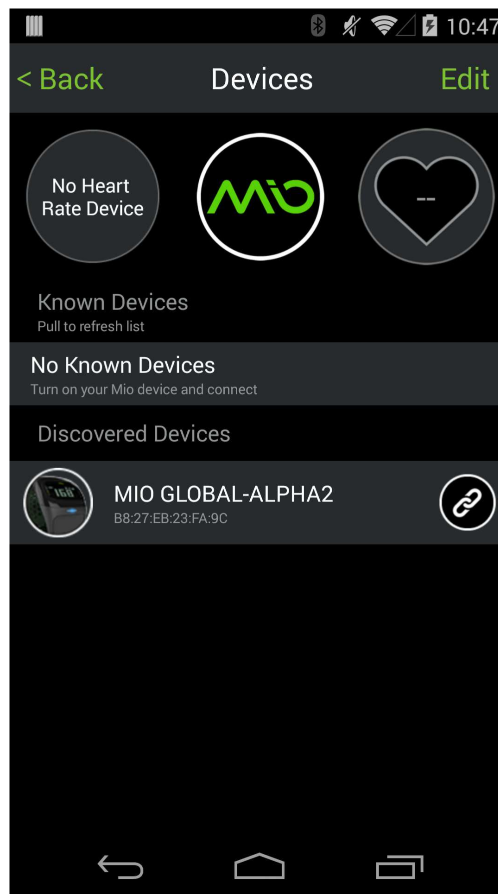


Figure 5.6: Faked Mio Alpha2 shows up in the original app

Mio Go it is most likely not that difficult, however, in other cases it might be very tedious. But if we looked further in the logs of the application we could see that the application had already unzipped the archive already, and written out its content to the file system. And as the archive was tainted, this taint was propagated to the unzipped file as well.

As we started the firmware update process, the unzipped binary file was read byte by byte, where these bytes got tainted as well, as they originated from a tainted file. From the bytes read a checksum was computed. Next, the application enabled notifications for the *DFU Control point* characteristic by writing to the *Client Characteristic Configuration*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

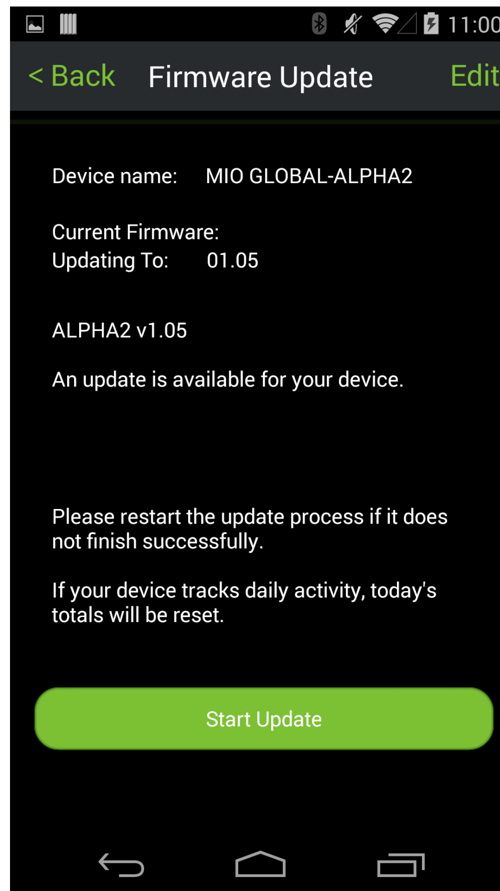


Figure 5.7: Faked Mio Alpha2 firmware update activity in Mio GO

descriptor. This implied that the firmware upgrade was in some way event driven and the server, in our case the Raspberry Pi, was expected to send some kind of a response as soon as some work had been done. Right after the notification had been enabled, the phone sent one byte (`0x01`) to the *DFU Control Point*, which corresponded with the opcode for starting the firmware upgrade process. Additionally, four bytes (`0x14 0xc0 0x01 0x00`) were sent to the *DFU Packet* characteristic. These bytes represented the size of the firmware update file in little endian format. As soon as the length had been received by the Raspberry Pi, the application seemingly stopped doing anything. This was a really good indication that our previous hypothesis was correct about the firmware update being event driven. When we took a close look at the decompiled code of the application, we could see that every change on the control point was parsed by the app, and depending on the "opcode" sent by the peripheral, different actions got executed by the application. In fact, had we written the following bytes `0x11 0x00 0x00 0x00 0x00 0x00 0x00 0x00`, the application would have sent us the next 20 bytes of the firmware update. `0x11` was the opcode to request new packages during firmware update,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

whereas the next four bytes indicated the number of received bytes, although seemingly it had no importance other than showing the progress of the firmware update on the GUI of the application. Even if receiving only forty bytes and sending back a number higher than that it would not result in sending the next package from that offset, as it was entirely handled by the application itself.

Unfortunately, apart from downloading and sending the firmware binary to the device, the application did not do anything else. For the Mio Go application the implemented framework helped to download and unpack the encrypted ZIP archive containing the firmware binary and to set up a device which could be identified by the application as a valid working device, which could support further analyses and reverse engineering the protocol used by the application and real devices. This support was especially helpful since this way both the device and the phone was controlled by the user, which in fact, provide more freedom and opportunities for an analyst.

5.2.4 Polar

At the end of 2018 Polar discontinued the support for devices running Android KitKat (4.4) and below, it means for our evaluation we used the last version (4.0.0) of Polar Flow which was compatible with TaintDroid.

Right after the start of the application, even without logging in, some UUIDs were picked up by the framework. Apart from some expected UUIDs the framework was able to identify a service UUID which belonged to the Polar company, as they seemingly had bought a license from the Bluetooth SIG. Moreover, some UUIDs had the same prefix which strongly suggested that they were not randomly generated, which made them significant for the analysis. The previously mentioned unidentified UUIDs can be seen in Table 5.7.

UUID
fb005c14-9815-d766-a528-32d54cf35530
fb005c16-02e7-f387-1cad-8acd2d8df0c8
fb005c17-02e7-f387-1cad-8acd2d8df0c8
fb005c18-02e7-f387-1cad-8acd2d8df0c8
fb005c19-02e7-f387-1cad-8acd2d8df0c8
fb005c51-02e7-f387-1cad-8acd2d8df0c8
fb005c52-02e7-f387-1cad-8acd2d8df0c8
fb005c53-02e7-f387-1cad-8acd2d8df0c8

Table 5.7: UUIDs with common prefix and in Polar Flow

In Table 5.7 apart from the first line every UUID only differs by one byte. This makes us believe that the first row represents the service UUID of Polar devices and the remaining UUIDs are characteristic UUIDs which belong to the previously mentioned service.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Connecting a device to Polar Flow, however, worked differently as anybody would have thought. According to the support page[3] of Polar, the new device should be registered first through a Windows or a Mac PC, and then the connection can be initiated from the device itself instead of from the phone. It means, that our framework will not be of any help without an actual polar device, except for the already described findings.

There is, however, another Polar application, namely Polar Beat, as described in Section 5.1. Similarly to the Polar Flow, Polar Beat dropped support for Android KitKat and below at the end of 2018. The last version, version 3.0.0, targeting older devices had a bug which resulted in continuous crashes upon the start of the application. Due to this, we had to revert to version 2.6.7 for this evaluation.

Through the three executions to search for a connectable device, without any faked or real device near to the test phone, the framework was able to leak 75, 55 and 73 uniquely occurring UUIDs respectively. The reason for the lower number of UUIDs for the second execution might have been that the application did not seem to have a timeout for the device search, and we probably canceled the search earlier than for the first and third executions. Correlating the UUIDs from the three executions resulted in a list of 34 UUIDs, which occurred throughout all three executions. From the 34 common UUIDs 17 were found in the framework's UUID-identification database. The remaining 17 unidentified UUIDs had the *fb005c* prefix and the same postfix as most of the previously mentioned UUIDs in the Polar Flow application. The remaining UUIDs, except for one, showed similarity in their prefixes, but their postfixes differed completely, which was atypical for UUIDs that belonged to a common service. The unidentified UUIDs can be seen in Table 5.8.

Similarly to the hypothesis with the unidentified UUIDs in the Polar Flow application, we suspected that the UUIDs with the prefix *fb005c* belonged together and they could be grouped into two services, namely service *fb005c20* and its characteristics *fb005c21-fb005c26* and service *fb005c50* and its characteristics *fb005c51-fb005c53*. The UUID *fffffde0* was found during independent evaluations with other applications as well, thus we suspect this UUID was most likely picked up from the environment and thus was irrelevant. As the framework also identified a UUID registered to the Polar company, it was likely that instead of *fb005c50* the registered UUID was the service and it was also a characteristic of that service. The same could apply for the UUIDs with *6217ff* prefix.

Next we tried to set up the Raspberry Pi in a way that Polar Beat identified it as an actual Polar H10 heart rate monitoring chest band. The first and obvious step was to start the GATT server on the Raspberry Pi with only a heart rate monitoring device, and it began advertising the heart rate monitoring service. As soon as the GATT server had been started, Polar Beat found the device and could even be paired with it. After the device had been paired, we could send out notifications on the heart rate measurement characteristic (Figure 5.8a), which was then displayed in the app (Figure 5.8b).

[3] `https://support.polar.com/en/support/M400/how_do_i_pair_my_polar_m400_with_the_polar_flow_app` Accessed: 22.05.2019

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID
6217ff49-ac7b-547e-eecf-016a06970ba9
6217ff4a-b07d-5deb-261e-2586752d942e
6217ff4b-fb31-1140-ad5a-a45545d7ecf3
6217ff4c-c8ec-b1fb-1380-3ad986708e2d
6217ff4d-91bb-91d0-7e2a-7cd3bda8a1f3
fb005c20-02e7-f387-1cad-8acd2d8df0c8
fb005c21-02e7-f387-1cad-8acd2d8df0c8
fb005c22-02e7-f387-1cad-8acd2d8df0c8
fb005c23-02e7-f387-1cad-8acd2d8df0c8
fb005c24-02e7-f387-1cad-8acd2d8df0c8
fb005c25-02e7-f387-1cad-8acd2d8df0c8
fb005c26-02e7-f387-1cad-8acd2d8df0c8
fb005c50-02e7-f387-1cad-8acd2d8df0c8
fb005c51-02e7-f387-1cad-8acd2d8df0c8
fb005c52-02e7-f387-1cad-8acd2d8df0c8
fb005c53-02e7-f387-1cad-8acd2d8df0c8
ffffde0-0000-1000-8000-00805f9b34fb

Table 5.8: Unidentified UUIDs in Polar Beat

Although it was good that we could connect a general heart rate monitoring device to Polar Beat, but this was supported out of the box, and our goal was to trigger a firmware update event, and so acquire the firmware files for the polar devices for further analyses. To achieve this we used the previously found UUIDs and searched for them in the disassembled APK to gather information about the grouping of the services and the characteristics and possibly about the meaning of these UUIDs. Table 5.9 shows the results of this search. As seemingly *ffffde0* was the only false positive UUID, we had a false discovery rate of less than 1%.

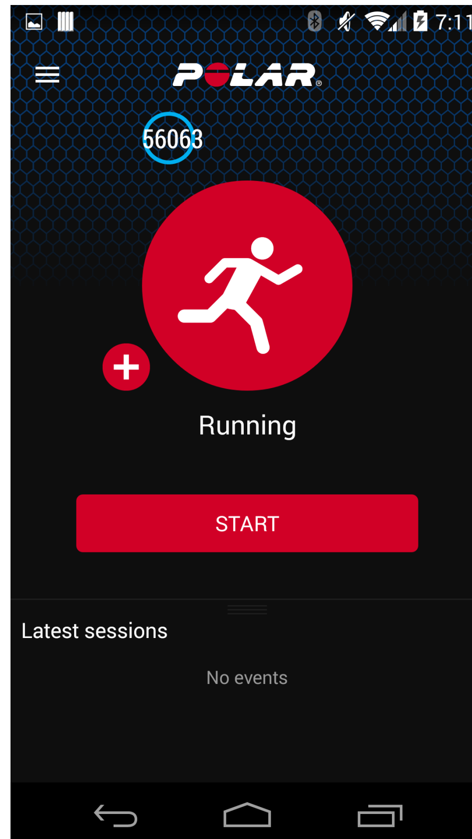
As soon as we had included the *Polar device control* service and its characteristics on the Raspberry Pi, the application showed that it had found an actual Polar device, and the Polar specific features are activated, see Figure 5.9a. In comparison the same features were not present when the device in use was not a Polar device, see Figure 5.9b.

Although the Raspberry Pi was recognized as a Polar H10, neither could we find the device on the profile page of our test account nor was every Polar specific features activated, like a fitness test. Furthermore, the application did not send a firmware update notification. Had we started the fitness test feature, the application would have showed a notification, that in order to use this feature we were required to pair the device to the phone. To do so we reconfigured the Raspberry Pi to be bondable and also not to require a PIN or a physical action for bonding. After restarting Polar Beat and re-initiating the pairing process, the Raspberry Pi received BLE traffic from the phone. As Figure

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
[GATT server]# Desc config Enabled for handle 0x002a: true
[GATT server]# Desc config already Enabled
[GATT server]# notify 0x0029 00 10 00 00
[GATT server]# notify 0x0029 00 11
[GATT server]# notify 0x0029 00 FF
[GATT server]# notify 0x0029 00 FF FF
[GATT server]# notify 0x0029 01 FF FF
[GATT server]# notify 0x0029 01 FF DB
```

(a) Sending out notifications on the Heart Rate Measurement



(b) Displaying the heart rate measurements

Figure 5.8: Polar Beat shows the heart rate measurements sent out by the Raspberry Pi

5.10 shows the application sets some notification flags and then writes some bytes to one of the Polar control characteristics. After the bytes had been written, the app went dormant which suggested that it was waiting for some kind of an answer.

Unfortunately, at this point our analysis got stuck, as we were not able to construct a valid response which would have been accepted by the application. Had one sent the same request to an original Polar H10, it would send back around 140 bytes with various

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

UUID	Meaning
6217ff49-ac7b-547e-eecf-016a06970ba9	H7 legacy settings
6217ff4a-b07d-5deb-261e-2586752d942e	H7 legacy settings
6217ff4b-fb31-1140-ad5a-a45545d7ecf3	PFC Service
6217ff4c-c8ec-b1fb-1380-3ad986708e2d	PFC Characteristic
6217ff4d-91bb-91d0-7e2a-7cd3bda8a1f3	PFC Characteristic
fb005c20-02e7-f387-1cad-8acd2d8df0c8	PSD Service
fb005c21-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
fb005c22-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
fb005c23-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
fb005c24-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
fb005c25-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
fb005c26-02e7-f387-1cad-8acd2d8df0c8	PSD Characteristic
0000feee-0000-1000-8000-00805f9b34fb	Polar device control
fb005c50-02e7-f387-1cad-8acd2d8df0c8	Device control char.
fb005c51-02e7-f387-1cad-8acd2d8df0c8	Device control char.
fb005c52-02e7-f387-1cad-8acd2d8df0c8	Device control char.
fb005c53-02e7-f387-1cad-8acd2d8df0c8	Device control char.

Table 5.9: Custom UUIDs in Polar Beat

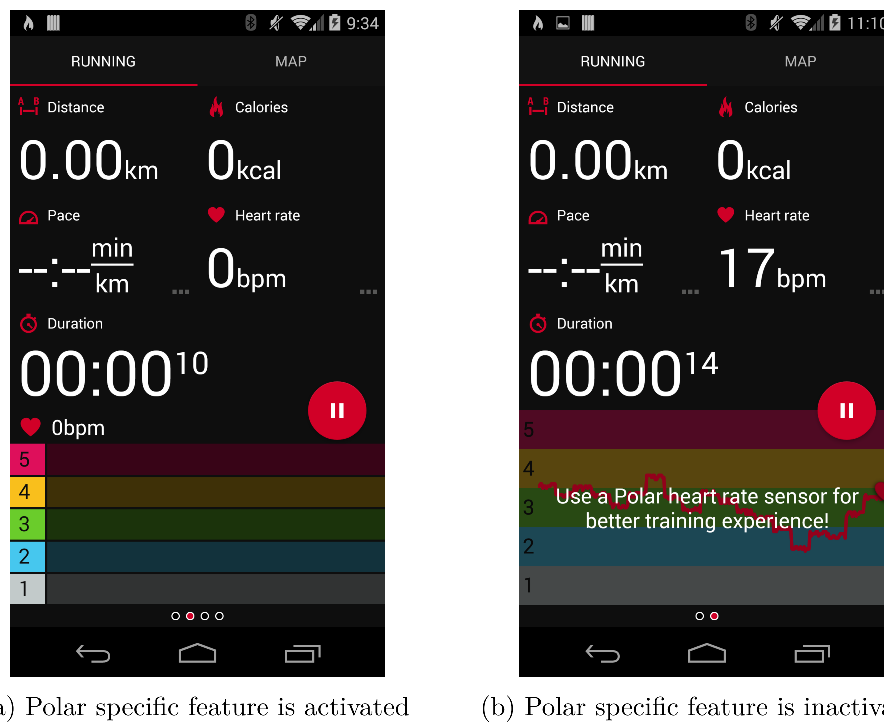
device specific information, like the model number, the model color, or the bootloader version. Among other things the device ID was also included, but even if we had been able to adapt the information to our chosen device ID, the app would not have been able to finish the pairing. According to the decompiled application, after receiving and parsing the response, Polar Beat would have sent the device information along with the user information to the backend server to presumably connect the device to the user, the pairing process would have be done.

Lastly, instead of using our Raspberry Pi to act as a real Polar device, we were going to start the evaluation with a real Polar H10 heart rate monitor.

Although we were able to pair the older version of Polar Beat to a newer device, namely a Polar H10 heart rate monitor, see Figure 5.11a, we were not able to initiate a firmware update, even though there was a newer firmware available which we had confirmed with another phone running the most current version of Polar Beat, see Figure 5.11b.

One possible reason for this might be that the API, on the backend through which the application could query if newer firmware is available, has been updated. And although the older version of the API can still be reached, no new information is served, at least not for devices which were announced after launching the new API.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Polar specific feature is activated (b) Polar specific feature is inactivated

Figure 5.9: Polar specific features are activated only in the presence of a polar device

5.2.5 Misfit

For this evaluation we used the most up-to-date version of Misfit as of writing this thesis, which was version 2.19.2.

After logging into our test account we were immediately greeted with a device selection page, which claimed that it could automatically identify nearby Misfit devices. This was an indication that the application listened to active advertisements, and most likely looked for specific advertised UUIDs. To find out which were the most likely candidates, we started our evaluation with three independent executions during which the leak UUIDs were collected. During these three executions 7, 6 and 6 unique UUIDs were found respectively, from which 2 were found in all three of them. These common UUIDs are shown in Table 5.10.

UUID
0b73b76a-cd65-4dc2-9585-aaa213320858
3dda0001-957f-7d4a-34a6-74696673696d

Table 5.10: Common UUIDs throughout three analysis executions in Misfit

As these UUIDs were not issued by the Bluetooth SIG the Raspberry Pi, and as a matter of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
Started listening on ATT channel. Waiting for connections
Connect from BC:F5:AC:60:B4:57
Running GATT server
[GATT server]# Desc config Enabled for handle 0x002a: true
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Desc config already Enabled
[GATT server]# Reading handle 0x00a9
[GATT server]# Reading handle 0x0089
[GATT server]# Reading handle 0x0049
[GATT server]# Reading handle 0x0015
[GATT server]# Reading handle 0x0013
[GATT server]# Reading handle 0x0011
[GATT server]# Reading handle 0x000f
[GATT server]# Reading handle 0x000d
[GATT server]# Reading handle 0x000b
[GATT server]# Reading handle 0x0009
[GATT server]# Received 18 bytes on handle 0x006c
[GATT server]# Data received (as string):
[GATT server]# Data received (as hex): 0x02 0x0f 0x00 0x08 0x00 0x12 0x0b 0x2f 0x44 0x45 0x56 0x49 0x43 0x45 0x2e 0x42 0x50 0x42
[GATT server]# Received 18 bytes on handle 0x006c
[GATT server]# Data received (as string):
[GATT server]# Data received (as hex): 0x02 0x0f 0x00 0x08 0x00 0x12 0x0b 0x2f 0x44 0x45 0x56 0x49 0x43 0x45 0x2e 0x42 0x50 0x42
[GATT server]#
```

Figure 5.10: GATT Server receives traffic after Polar Beat bonds to the Raspberry Pi

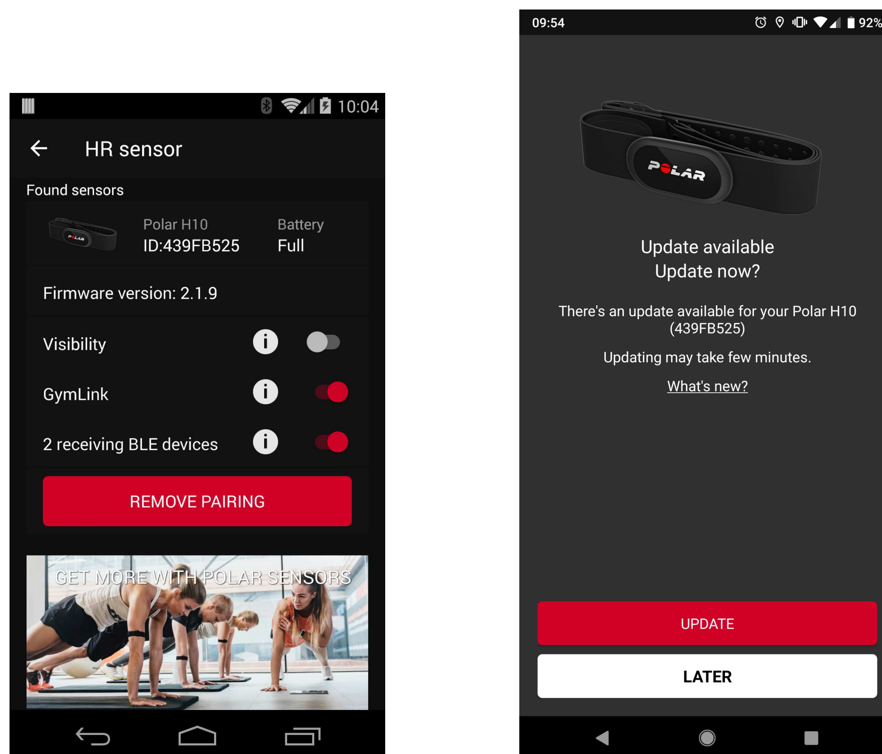
fact every Bluetooth Low Energy device below Bluetooth 5.0, could only advertise one such UUID at a time. The reason for this is that the size of the advertising package is limited to 31 bytes, and such non-standard UUIDs, or more precisely UUIDs which do not share the same postfix as the base UUID (`00000000-0000-1000-8000-00805f9b34fb`) defined by the Bluetooth SIG, cannot be shortened to their 16-bit format. It means the whole 128-bit UUID must be advertised, which alone needs 18 bytes (16 bytes UUID, 1 byte field identifier and 1 byte size). Furthermore, we must set 3 more bytes (1 byte field identifier, 1 byte size and 1 byte data) for the connection flags data type, which cannot be omitted, as we explicitly set the *BR/EDR Not Supported* flag. The reason for this is discussed later in Section 6.2.4.

To contradict our hypothesis, that the manufacturer data is not relevant during advertisements, the Misfit application uses the manufacturer's specific field. Tracing the usage of this field makes it clear that the serial number of the device is stored in that field, and it is expected to be at least ten bytes long. Adding the new manufacturer's specific field to the advertisement package requires 14 bytes (10 bytes data, 1 byte size, 1byte field identifier and 2 additional bytes used as a company identifier code). This, however, would not fit into the advertisement package. Fortunately, we can specify the content of the scan response package which is only sent back to the client if the server were scanned actively. It gives us 31 additional bytes to use.

To summarize, we set the advertisement to advertise the above mentioned UUID and the BLE flags and we provide both the device's name and the serial number through the scan response package.

Starting the scanning process in the application lead to some SSL exceptions due to probably unverifiable hosts. When we took a close look inside the application we could see that it tried to upload some files to `https://data.misfit.com`. Tracing back

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub



(a) Polar H10 is paired to an older version of Polar Beat on TaintDroid

(b) Polar H10 firmware update available through newer version of Polar Beat

Figure 5.11: Availability of a firmware update depends on the version of the application

from which path the file had been read lead us to several Base64-encoded files, which after decoding seemed to be encrypted. Although the encryption password is randomly generated, it was stored in the shared preferences of the app which was an unencrypted XML file. Decryption of the found files gave back the scan results in JSON-format. A shortened version of one of these files can be seen in the appendix in Listing 8.1

After sending the same request to the previously found host with the unencrypted payload from outside of the app, the server sent back a JSON with a session ID and a message stating that the executed action had been successful. Due to time constraints at this point, we discontinued our evaluation with the Misfit application, and we were going to analyze what had caused the problem with the SSL handshake in detail.

5.3 Result correlation

Table 5.11 shows the summarized results of the tests we conducted on the applications of the sample set.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Application name	Identify relevant UUIDs	UUID leak	Fake presence	Connect faked device	Pair faked device	Dump firmware (fake device)	Dump firmware (real device)
	Feature test						
MIFit	●	●	●	●	◐	-	-
Fitbit	●	●	●	●	◐	-	●
MioGo	●	●	●	●	●	●	X
Polar	●	●	●	●	◐	-	-
Misfit	●	●	●	-	X	X	X

●=works completely; ◐=partially works; -=doesn't work; X=not applicable;

Table 5.11: Test results for each application in the sample set

As we can see in every application we were able to leak and identify the UUIDs which were relevant for the applications to communicate with the specific devices. With the help of these UUIDs we were able to configure a Bluetooth Low Energy server for each application which was recognized as if it were a device which was manufactured by the app manufacturer. Except for one application, the falsified device was connected and the pairing process has been initiated. As we could verify it with the help of the Android logs, the last application was not able to connect to its backend server, which caused the problems during the connection.

Although connecting to a device was mostly not a problem, pairing the device to the user account worked only partially for most of the applications in the sample set. The reasons for this varied from requiring some sort of cryptographic challenge response for which we did not have the key, or the application arrived in a state that it thought the pairing had been successful, however, the pairing request either never arrived at the backend, or it was declined. This resulted in a state where the application temporarily showed that the pairing was successful, but after restarting or refreshing the application, it pulled the data from the backend and reverted to a state where the pairing never happened.

Initiating a firmware update and dumping the packages worked only for the Mio Go application, which was the only application in our sample set, where the pairing was completely successful. For the remaining applications due to an error at the previous steps, the device was not connected to the user account, and thus never queried if there were new firmwares available.

For the last test we required working physical devices which we only had three (Fitbit, MI Fit and Polar) out of the five sample applications, thus for the remaining two (Misfit and Mio Go) this test was not applicable. Dumping the firmware with the help of a real device worked only with Fitbit, from which only the first few thousand bytes were dumped due to a memory corruption bug. For MI Fit this last test did not work, as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

opposed to our hypothesis - that is the firmware updates would be downloaded from the Internet - the firmware updates were distributed as application assets, thus they were not tainted and our firmware did not recognize them as relevant traffic. And lastly, Polar most likely changed the backend API and although we verified it with a newer version of Polar Beat, that an update existed for our device, the old version of the application could not find this.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Discussion and Results

In the following sections we compare our framework to similar existing solutions, then we discuss the limitations and problems we encountered during the implementation, and finally we discuss the results of the system evaluation.

6.1 Comparison to Existing Solutions

In this section we are going to compare the implemented framework to currently existing solutions. Nowadays only two methods are comparable to our solution, namely IoTFuzzer [40] implemented by Chen et. al. and manual reversing.

6.1.1 IoTFuzzer

IoTFuzzer is a framework for automated black box protocol-guided fuzzing of IoT device firmwares. Similarly to our intuition Chen et. al. found that a big portion of modern embedded devices are controlled by the user's smartphone. As extracting a firmware and running it in an emulator can be very cumbersome, they proposed that the information could be gathered from the smartphone application and later this information could be used to fuzz the firmware on the device without the need to extract it. One of the main differences between our framework and IoTFuzzer is that our framework focuses on Bluetooth Low Energy based protocols whereas IoTFuzzer focuses on devices which use WiFi to connect to the phone.

IoTFuzzer first starts a UI analysis in order to identify actions which would trigger sending network packages. As our system is designed in a human-assisted manner, we do not prepare such an analysis and let the user trigger the needed actions.

Similarly to our approach IoTFuzzer uses a modified version of TaintDroid. Unlike our implementation, IoTFuzzer prevents the propagation of taints in some cases like

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

encryption, as their goal is to mutate the output before the encryption happens, which means they would not gather any more useful information if the encrypted messages got tainted. But just like us, IoTFuzzer extended both the taint sources and the sinks to serve their purpose.

After running the data-flow analysis the steps of the two frameworks differ extensively. IoTFuzzer, being a fuzzer, begins to mutate the identified protocol fields during runtime, and at the same time the response of the device is monitored, either with a heartbeat mechanism (for UDP-based communication) or by identifying the communication interruption between phone and device (TCP-based communication). On the other hand, we try to reach a point where the application thinks it should push the new firmware image to the device, which point we intercept the communication, dump the firmware and also try to simulate the presence of a real device to achieve the previous goal only without the need of a real device.

6.1.2 Manual Reversing

Section 2.4 illustrates the general approach during manual reverse engineering. The key difference between manual reversing and our presented solution is the automation of reconnaissance, the accumulation and the presentation of information.

In detail, the framework collects and identifies the used Bluetooth UUIDs and sorts out the randomly generated ones. Additionally, if data is downloaded from the Internet and is sent to a Bluetooth device, this data will be dumped by the framework, whereas in case of manual reversing this task would be relatively cumbersome if the file is never written out to disk or if it is deleted after usage.

Furthermore, the framework includes components that try to emulate the presence of real Bluetooth devices, which possibly enables the analysis of the used protocol and dumping the firmware even if the real device is unavailable for the person conducting the analysis.

6.2 Limitations

This section outlines the problems which were encountered during the implementation of the system, as well as their attempted solutions, and the limitations the unsolved problems caused.

6.2.1 Tainting in Android

Two dynamic tainting frameworks for Android have been analyzed to build upon, TaintDroid and TaintART. As it was mentioned before the runtime system for Android has been changed with Android 5.0 from Dalvik to ART. TaintART is written for this newer environment and currently supports Android 5.0 and Android 6.0. On the other hand TaintDroid aims at supporting the older environment. As of this writing TaintDroid supports Android 2.1, 2.3, 4.1 and 4.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Although TaintART supports the newest versions of Android, and thus more recent applications can be analyzed, unfortunately, it lacks examples how one could interact with the tainting system. However, the developers of TaintDroid published not only the code for the tainting system, but also how they instrumented several parts of the system, which was a good source of inspiration for extending it.

6.2.2 Build System Limitations

Generally the Android build system only supports building Android for a real device or for the Android Emulator. As the aim of this project is to track Bluetooth traffic leaving and arriving Android, the emulator is a nonviable option, as it lacks hardware virtualization for several components, with Bluetooth among them. [77] Without the virtualized hardware the complete Bluetooth functions turned off in the emulator.

The other option, building for a real device, has its own limitations as well. The person who uses the system has to have the hardware, onto which the built ROM has to be flashed, and the appropriate version of the device drivers needs to be found on the Internet and compiled into the ROM as well.

6.2.3 Android in a VM

To overcome the limitations of Android build system, the authors planned to start Android inside a virtual machine (VM). Although the build system has a target for VirtualBox-images, the built images were not bootable. More precisely, the bootloader was unable to start or could not load its later stages. Unfortunately, there was no way to get more detailed output from the system, and it also seemed to be unresponsive to inputs. Furthermore, according to forum posts this target seems to be unmaintained and is advised not to be used.

There is, however, a project called Android-x86, its goal is to produce bootable Android ISOs. As the project is built on the x86 architecture inside the VM, some changes were imminent. It provides the ISOs and the sources to almost all released versions of Android. Their implementation for JellyBean is based on Android 4.3.1, whereas TaintDroid uses Android 4.3.0. Although one would think the difference is only a minor release, combining the sources of TaintDroid and Android-x86 it resulted in several compiler errors. To address this issue TaintDroid needed to be updated to the codebase of Android 4.3.1.

After updating to Android 4.3.1 TaintDroid was able to boot inside a VM, however, some issues with Bluetooth have been found. Even though BR/EDR was working as expected, the BLE subsystem was somehow not working. As it turned out it was a known issue and was fixed in the Android-x86 with Android 4.4.4 as they changed their Bluetooth driver to BlueZ. As a solution, TaintDroid needed to be updated again.

BlueZ Low Energy, among other options, features the possibility to use encryption during transmission. To minimize the codebase and to reuse code, BlueZ neither implements the cryptographic functions itself nor uses libraries built for this, but accesses the crypto API

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of the Linux kernel. The communication with the API and a process in the userspace work with specialized sockets. For example: if one would like to encrypt something with AES-CBC, s/he would create a socket with type `skcipher` and name `cbc(aes)`. The former would activate the symmetric crypto algorithms and the latter would activate the specific algorithm if it is available. After binding to this socket and setting the key, one could encrypt and decrypt data by using the `sendmsg()` and `recv()` system calls known from network sockets.

Albeit the configuration of the kernel allowed accessing the crypto API from userspace, and the used algorithms were correctly loaded as modules, the API was not reachable for BlueZ therefore the BLE subsystem could not be initialized. Despite several kernel and driver replacements the problem still existed, which was solved in Android-x86 with Android 5.0.

Apart from having problems with Bluetooth, the Android VMs were unreliable, as they crashed very often without having a real cause. The authors found no indication for these crashes either in the logs of the VM itself or in the logs of Android.

As a conclusion, the authors decided to flash TaintDroid onto a real device instead of using a VM despite all the aforementioned inconveniences.

6.2.4 Issues with the Generic GATT Server

During the implementations of the GATT server on the Raspberry Pi, the authors encountered two main issues.

The first issue was that before Android 5.0 during the initial connection there was no option to specify if BR/EDR or BLE should be used. This behavior of Android resulted that the particular Broadcom Chip inside the test phone tried to connect in dual mode. As the advertiser of the services (a Raspberry Pi) offered both BR/EDR and BLE, the chip chose the one which could be initialized faster. Through this the situation arose in which the test-application could not communicate through BLE reliably and the initial tests failed. The solution was to turn off Bluetooth Classic on the advertiser.

The second issue was that although the services and characteristics were configured correctly and the advertisement worked well right after the connection was established by the applications, the connection was aborted by the client. Later we found, that although the BLE objects were in place no handler method was bound to them and thus the server could not send back the expected responses as the client tried to interact with it. This, of course, resulted in an exception which was handled by aborting the connection. Its solution was to implement some generic callback methods for the server, which sent back the minimal required packages.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

6.3 Discussion

Nowadays a lot of small embedded devices, also called as the Internet of Things (IoT) are brought to the market. A common feature of these devices, regardless of their application, is that they are all equipped with some form of communication interface, like WiFi or Bluetooth. Bluetooth Low Energy (BLE) is a very popular choice for these devices, since the name suggests it requires minimal energy input, which results in a longer battery life.

The main issue with BLE is that devices cannot directly communicate with the backend system where the data aggregation and evaluation is realized. To overcome this issue, a popular choice of manufacturers is to use the user's smartphone to act as a middle man between the backend servers and the IoT device. It means the application on the phone can be a rich source of information even if the goal at the end is to analyze the firmware of the device.

One of our main goals was to make it possible to extract the firmware of the IoT-device, if possible in its decrypted form, from the application with minimal investment of cumbersome reverse engineering of obfuscated Android applications. Our other objective was to try to create a generic re-configurable BLE device which could act as a real device and can convince the application on the phone, that it is a real device, and this way provide an opportunity to analyze the protocol without the need to possess the real device.

To fake a Bluetooth Low Energy device a list of the required BLE service and characteristic UUIDs were needed, since the app on the phone expected them to be present. Our framework was able to extract this information from every application we selected to test the framework with, without disabling the application itself. With the help of the information provided by our framework we were able to create a device configuration for four out of five applications which lead to at least a partial recognition of our general purpose BLE device as an actual fitness tracker. By partial recognition we mean that the application initiated a connection and sent data to the selected characteristics, and reacted as if we had written to these selected characteristics. In case of one of the test applications we were able to bring the application into a state where it started the firmware update process and sent the firmware image to our "rouge" device.

The implication of these results is that one can harvest the configuration for a valid BLE fitness tracker with only minimal manual labor without the need to possess an actual tracker.

Out of the three real devices we had to run evaluations, we only managed to extract parts of a firmware for just one device. For Fitbit we managed to dump the firmware update traffic during transmission, however, due to a bug either in our extensions or in TaintDroid the transmission got interrupted, but nonetheless we were able to show that our system works. For the remaining two devices, the reason for not being able to dump the firmware files were distinct. As we mentioned, MI Fit actually releases the firmware update files as part of the application and thus they are written on the file

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

system during the installation which means, that they are never tainted. However, the question arises whether it would still be true if the app were installed through Google Play instead of through the Android debug interface (*adb*), thus being downloaded from the Internet and thus being tainted. This will be evaluated in the future. As for Polar Beat, the application has never shown any available update despite the fact that there was one as we confirmed it with a newer phone. We suspect however, that our method would have worked if the backend had answered the query correctly.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Summary and Future Work

This thesis presents a new analysis framework which can be used to extract Bluetooth Low Energy related information from an Android application. This information is then used in the second part of the framework which is a generic BLE server with an interface which can be used to reconfigure the server in an uncomplicated manner. The generic BLE server can be configured to include any number of services and characteristics, which makes it possible to simulate the presence of a real BLE device to which the application can connect on the smartphone. Through this both the application and the device are under control of the analyst, which provides a straightforward way to analyze the protocol used to communicate between phone and device. Furthermore the presented framework writes out any data which was downloaded from the Internet and is sent out through the Bluetooth interface providing a new method to extract the firmware of the device without the need of complicated firmware extraction methods.

The implemented framework was tested with three real life fitness tracking applications. With the extracted information for the BLE device, we were able to infer the needed configuration and to configure the generic BLE server in a way the application recognized it as its own device. Four of the applications were able to connect to the configured device, and the pairing to the test account worked partially for these four applications as well. In one case we were able to dump the firmware update with the use of the generic BLE server.

The current system has still potential for improvements, as several tasks are left to be executed manually by the user conducting the analysis. One such task would be to automatically identify callback methods, which handle the asynchronous communication between the phone and the device. Identifying these methods is crucial in the BLE analysis, as these methods represent the communication logic of the application.

If one were more interested in analyzing the protocol spoken by the phone and the device, the system should be extended by hooking into the methods which are responsible for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

reading and writing BLE characteristics. This would most likely mean that a new taint source would have to be defined, incoming traffic through the Bluetooth interface, so the analysis could reveal what happens with the answer provided by the device. Additional warnings could be printed as well with trace logs, to further narrow the search space inside the application in the hunt for finding the classes responsible for handling the BLE communication.

In case of faking the device, it would make sense to extend a system in a way, that it could autonomously try to guess valid responses expected by the client to arrive. This extension would eliminate the need to reverse engineer the protocol used by the phone and the real device, thus it would save time.

The most urgent task regarding future improvements would be to port the system to an ART based tainting framework. Android 4.4 is less and less supported by newer versions of applications, which both delivers potentially biased results as newer versions of the application might use another protocol and potentially prevents analyzing new devices, as they might not be supported by the older version of the application.

Although Enck et. al. conducted a through performance and memory overhead analysis of TaintDroid [31], due to additional taint sources, sinks and tags we plan to rerun their tests in order to verify, if our changes caused any difference in performance or memory usage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 8

Appendix

```
1  {
2    "device_model": "LGE_AOSP on HammerHead",
3    "platform": "Android",
4    "end_at": 1557306854,
5    "events": [
6      {
7        "requestStarted": {
8          "value": {
9            "callback": "21be8548"
10         },
11         "timestamp": 1557306834.675
12       },
13       "event": "startScanning"
14     },
15     {
16       "responseFinished": {
17         "value": {
18           "data": "02010611066D6973666974A6344A7D7F950100
               DA3D0DFFDF00423030525A303030303106085368696E
               650000000000000000000000000000000000000000",
19           "deviceName": "Shine",
20           "serialNumber": "B00RZ00001",
21           "address": "B8:27:EB:23:FA:9C"
22         },
23         "timestamp": 1557306835.125,
24         "result": 0
25       },
26       "event": "scanResult"
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
    },
    {
      "responseFinished": {
        "value": {
          "data": "1AFF4C00021550765CB7D9EA4E2199A4FA8796
             13A49231076EA6CE0000000000000000000000000000
             000000000000000000000000000000000000000000",
          "address": "B0:52:16:D3:4B:B6"
        },
        "timestamp": 1557306835.19,
        "result": 0
      },
      "event": "scanResult"
    },
    {
      "requestStarted": {
        "value": {
          "callback": "21be8548"
        },
        "timestamp": 1557306854.709
      },
      "event": "stopScanning"
    }
  ],
  "sdk_version": "2.6.1-misfit-release",
  "start_at": 1557306834,
  "user_id": "5ccbe2dce4b08512a245db43",
  "system_version": "4.4.4"
}
```


Listing 8.1: Decrypted JSON used by Misfit

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

List of Figures

2.1 Bluetooth-stack [7] 6
2.2 Possible controller configurations[7] 7
2.3 Worldwide mobile OS market share [12] 11
2.4 Android-stack [13] 12
2.5 Layers of the Hardware Abstraction Layer [16] 13
2.6 Output of apktool 18

3.1 TaintDroid propagation rules [31] 29
3.2 NDroid architecture [76] 29
3.3 NDroid propagation rules [76] 30
3.4 TaintART propagation rules [31] 30

4.1 Architecture of the implemented system 37

5.1 First package after connection 42
5.2 GATT-Server disconnection during Fitbit simulation 47
5.3 GATT-Server receives the first package from Fitbit 47
5.4 Firmware update progress on a Fitbit One 48
5.5 Entropy of the Fitbit firmware image 49
5.6 Faked Mio Alpha2 shows up in the original app 52
5.7 Faked Mio Alpha2 firmware update activity in Mio GO 53
5.8 Polar Beat shows the heart rate measurements sent out by the Raspberry Pi 57
5.9 Polar specific features are activated only in the presence of a polar device 59
5.10 GATT Server receives traffic after Polar Beat bonds to the Raspberry Pi 60
5.11 Availability of a firmware update depends on the version of the application 61

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

5.1 Similar UUIDs throughout three analysis executions and their meaning (entries with * were categorized manually) . . . 42
5.2 Similar UUIDs throughout three analysis executions and their meaning for the Fitbit application . . . 45
5.3 UUIDs which were found in three executions of the Fitbit application although they were categorized manually . . . 46
5.4 Unknown UUIDs with high similarity found during the execution of Mio GO 50
5.5 Common UUIDs along three independent devices search in Mio GO (rows with * were identified manually) . . . 51
5.6 Additional UUIDs which are never used in Mio GO . . . 52
5.7 UUIDs with common prefix and in Polar Flow . . . 54
5.8 Unidentified UUIDs in Polar Beat . . . 56
5.9 Custom UUIDs in Polar Beat . . . 58
5.10 Common UUIDs throughout three analysis executions in Misfit . . . 59
5.11 Test results for each application in the sample set . . . 62

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Listings

2.1 Search for an application on the phone 18
4.1 UUID regex 31
4.2 Don't taint traffic for specific domains 33
4.3 Leaking UUIDs as soon as they are created 34
5.1 Pattern to access a specific characteristic 42
8.1 Decrypted JSON used by Misfit 73

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Statista, "Internet of things (iot) connected devices installed base worldwide from 2015 to 2025 (in billions)," https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, online; Accessed: 04.03.2019.

[2] C. W. Axelrod, "Enforcing security, safety and privacy for the internet of things," in *Systems, Applications and Technology Conference (LISAT), 2015 IEEE Long Island*. IEEE, 2015, pp. 1–6.

[3] Österreichischer Nationalrat, "Medizinproduktegesetz," 1996, version 25.01.2018.

[4] U.S. Food & Drug Administration, "Firmware update to address cybersecurity vulnerabilities identified in abbott's (formerly st. jude medical's) implantable cardiac pacemakers: Fda safety communication," https://www.fda.gov/MedicalDevices/Safety/AlertsandNotices/ucm573669.htm, online; Accessed: 02.11.2017.

[5] J. Sametinger, J. Rozenblit, R. Lysecky, and P. Ott, "Security challenges for medical devices," *Commun. ACM*, vol. 58, no. 4, pp. 74–82, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2667218

[6] Bluetooth SIG. (2018) Bluetooth market update 2018. Accessed: 2019-01-28. [Online]. Available: https://www.bluetooth.com/markets/market-report

[7] ——, "Core specification v5.0," 2018, accessed: 2019-01-18. [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification

[8] Silicon labs, "Ug103.14: Bluetooth le fundamentals," accessed: 2019-01-18. [Online]. Available: https://www.silabs.com/documents/login/user-guides/ug103-14-fundamentals-ble.pdf

[9] Bluetooth SIG Medical Working Group, "Health thermometer profile v10.0," 2011, accessed: 2019-01-18. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=238687

[10] Bluetooth SIG GPA Working Group, "Device information service v11.0," 2011, accessed: 2019-01-18. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=244369

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[11] Bluetooth SIG Medical Working Group, "Health thermometer service v10.0," 2011, accessed: 2019-01-18. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=238688

[12] Statcounter, "Mobile operation system market share worldwide," accessed: 2019-05-17. [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide

[13] Google, "The android source code," accessed: 2019-01-18. [Online]. Available: https://source.android.com/images/android_framework_details.png

[14] F. Maker and Y.-H. Chan, "A survey on android vs. linux," *University of California*, pp. 1–10, 2009.

[15] Google, "Selinux concepts," accessed: 2019-01-18. [Online]. Available: https://source.android.com/security/selinux/concepts

[16] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing.* " O'Reilly Media, Inc.", 2013.

[17] W. Stallings, *Operating systems: internals and design principles.* Boston: Prentice Hall,, 2012.

[18] D. Bornstein, "Dalvik vm internals," accessed: 2019-01-18. [Online]. Available: http://fiona.dmcs.pl/podyplomowe_smtm/smob3/Presentation-Of-Dalvik-VM-Internals.pdf

[19] Google, "Art and dalvik," accessed: 2019-01-18. [Online]. Available: https://source.android.com/devices/tech/dalvik

[20] ——, "Application fundamentals," accessed: 2019-05-24. [Online]. Available: https://developer.android.com/guide/components/fundamentals

[21] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[22] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM, 2007, pp. 196–206.

[23] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on.* IEEE, 2010, pp. 317–331.

[24] A. C. Myers and A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 1999, pp. 228–241.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[25] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 2012, pp. 3–8.

[26] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *Software Engineering (WCSE), 2012 Third World Congress on*. IEEE, 2012, pp. 101–104.

[27] 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," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[28] F. Pottier and V. Simonet, "Information flow inference for ml," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 1, pp. 117–158, 2003.

[29] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, 2014, pp. 1–6.

[30] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Highly precise taint analysis for android applications," 2013.

[31] 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 Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[32] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 331–342.

[33] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *NDSS*, 2014.

[34] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *NDSS*, 2015.

[35] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 437–448.

[36] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *NDSS*, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[37] K. Koscher, T. Kohno, and D. Molnar, "Surrogates: Enabling near-real-time dynamic analyses of embedded systems." in *WOOT*, 2015.

[38] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution." in *USENIX Security Symposium*, 2013, pp. 463–478.

[39] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 337–342.

[40] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing."

[41] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[42] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation."

[43] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[44] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *International Conference on Information and Communications Security*. Springer, 2008, pp. 238–255.

[45] J. Ming, M. Pan, and D. Gao, "ibinhunt: Binary hunting with inter-procedural control flow," in *International Conference on Information Security and Cryptology*. Springer, 2012, pp. 92–109.

[46] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Security Symposium*, 2014, pp. 95–110.

[47] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 406–415.

[48] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[49] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 480–491.

[50] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[51] H. Flake, "Structural comparison of executable objects," in *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*. Citeseer, 2004, pp. 161–174.

[52] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," vol. 5, 01 2005.

[53] S. L. Thomas, F. D. Garcia, and T. Chothia, "Humidify: a tool for hidden functionality detection in firmware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 279–300.

[54] M. M. Hossain, M. Fotouhi, and R. Hasan, "Towards an analysis of security issues, challenges, and open problems in the internet of things," in *Services (SERVICES), 2015 IEEE World Congress on*. IEEE, 2015, pp. 21–28.

[55] D. Halperin, T. S. Heydt-Benjamin, K. Fu, T. Kohno, and W. H. Maisel, "Security and privacy for implantable medical devices," *IEEE pervasive computing*, vol. 7, no. 1, 2008.

[56] S. Hanna, R. Rolles, A. Molina-Markham, P. Poosankam, J. Blocki, K. Fu, and D. Song, "Take two software updates and see me in the morning: The case for software security evaluations of medical devices." in *HealthSec*, 2011.

[57] C. Folk, D. Hurley, W. K. Kaplow, and J. F. Payne, "The security implications of the internet of things," *Fairfax: AFCEA International Cyber Committee*, 2015.

[58] Google, "apkanalyzer," accessed: 2019-03-07. [Online]. Available: https://developer.android.com/studio/command-line/apkanalyzer

[59] R. Wiśniewski and C. Tumbleson, "Apktool," accessed: 2019-03-07. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[60] B. Pan, "dex2jar," accessed: 2019-03-07. [Online]. Available: https://github.com/pxb1988/dex2jar

[61] D. Octeau, W. Enck, and P. McDaniel, "The ded decompiler," *Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0140-2010*, 2010.

[62] E. Dupuy, "jd," accessed: 2019-03-07. [Online]. Available: http://java-decompiler.github.io/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

[63] Skylot, "jadx," accessed: 2019-03-07. [Online]. Available: https://github.com/skylot/jadx

[64] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Proc. of Black Hat Abu Dhabi*, pp. 77–101, 2011.

[65] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 343–355.

[66] R. Baumann, M. Protsenko, and T. Müller, "Anti-proguard: Towards automated deobfuscation of android apps," in *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*. ACM, 2017, pp. 7–12.

[67] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android," Tech. Rep., 2009.

[68] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[69] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[70] IBM T.J. Watson Research Center, "T.j. watson libraries for analysis," accessed: 2019-02-06. [Online]. Available: http://wala.sourceforge.net

[71] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[72] X. Cui, J. Wang, L. C. Hui, Z. Xie, T. Zeng, and S.-M. Yiu, "Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 2015, p. 25.

[73] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[74] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, p. 14, 2018.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[75] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, 2015, p. 110.

[76] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2014, pp. 180–191.

[77] Google, "Android emulator limitations," accessed: 2019-05-17. [Online]. Available: https://developer.android.com/studio/run/emulator#limitations

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub