TU WIEN Informatics

# Large-scale Static Analysis of PII Leakage in IoT Companion Apps

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**David Schmidt, BSc**
Matrikelnummer 01525460

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn Martina Lindorfer
Mitwirkung: Jakob Bleier, BSc BSc MSc

Wien, 20. Mai 2021

David Schmidt | Martina Lindorfer

TU WIEN Informatics

# Large-scale Static Analysis of PII Leakage in IoT Companion Apps

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**David Schmidt, BSc**
Registration Number 01525460

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn Martina Lindorfer
Assistance: Jakob Bleier, BSc BSc MSc

Vienna, 20th May, 2021

David Schmidt | Martina Lindorfer

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

# Erklärung zur Verfassung der Arbeit

David Schmidt, BSc

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, 20. Mai 2021

David Schmidt

# Acknowledgements

I want to thank all the people who supported me. Without you, it would never be possible for me to finish my studies in the way I did.

Also, I want to thank my parents for making all this possible in the first place.

# Kurzfassung

Oft sind Internet of Things (IoT) Geräte aufgrund von Sicherheits- und Datenschutzproblemen in den Nachrichten. Eine groß angelegte Analyse ist eine Möglichkeit, die aktuelle Situation zu verbessern. Forscher und Hersteller können eine solche Analyse dazu verwenden, Schwachstellen zu finden, diese zu melden und zu beheben, bevor sie ausgenützt werden. Jedoch müssen, um eine derartige Analyse durchzuführen zu können, zunächst zwei Herausforderungen überwunden werden. Zunächst, sind IoT Geräte verschieden in Bezug auf ihre Soft- und Hardware, was ein einheitliches Vorgehen schwierig macht. Außerdem sind Analysen oft mit hohen Kosten verbunden, wenn physische Geräte beim gewählten Analzseansatz benötigt werden.

Um die beiden Herausforderungen zu umgehen, haben wir einen statischen Analyseansatz für IoT Companion Apps entwickelt. Bei Companion Apps handelt es sich um Smartphone Apps, mit denen es möglich ist mit den physischen Geräten zu interagieren. Wir haben uns dabei auf zwei Aspekte von Companion Apps fokussiert, die sie von anderen Apps unterscheiden. Nämlich, die Kommunikation über das lokale Netzwerk und die verwendeten Protokolle. Für unsere Arbeit haben wir zwei Analysetechniken gewählt, Taint Analyse und Value Set Analyse. Letztere verwenden wir dazu, URLs zu extrahieren und damit lokale Kommunikation zu finden. Für die Arbeit haben wir insgesamt 124 Companion Apps analysiert. Wir zeigen, welche Informationen dadurch bereits von den rekonstruierten Endpunkten gewonnen werden. Außerdem zeigen wir von zwei Companion Apps die Datenflüsse im Detail, diese beinhalten Privatsphäre und Sicherheits Bedrohungen. Wir machen damit einen Schritt in Richtung einer groß angelegten Analyse, um Datenschutzprobleme in Companion Apps zu finden.

# Abstract

Security and privacy problems of smart devices are often reported in the news. One possibility to improve the current situation are large-scale analyzes. Researchers and manufacturers can use such analysis to detect weaknesses, report them, and fix them before they get misused. However, to be able to perform a large-scale analysis, two difficulties need to be overcome. First, the diversity regarding software and hardware of smart devices makes a general approach difficult. Second, analyzes are often associated with high costs if physical devices are needed for the selected approach.

We developed a static analysis approach for Internet of Things (IoT) companion applications (apps) to circumvent those difficulties. Companion apps are mobile apps, allowing their users to interact with smart devices. We focused on two aspects of companion apps that distinguish them from other applications: the communication over the local network and the used protocols. For this thesis, we use two analysis techniques to collect further information about the devices: taint analysis and value set analysis. We have chosen the latter for reconstructing URLs called by the applications and thereby detecting local communication. In this thesis, we analyzed in total 124 companion apps with our approach. We show the information obtained by the reconstructed endpoints. Furthermore, we present the flows found in two companion apps in detail, which contain threats to user's security and privacy. Overall, we make one step towards large-scale analysis of personally identifiable information (PII) leakage in IoT companion apps.

# Contents

CHAPTER 1

# Introduction

## 1.1 Problem Statement

Physical objects are equipped with sensors and connected to networks to make them smart. Objects attached with computational power are called "smart things." The amount of smart devices in private homes is increasing. Nowadays, even whole buildings are equipped with smart devices.

With this development, the number of smart devices is growing from 22 billion in 2018 to 50 billion in 2030, according to a forecast.[1]

Devices communicate with each other to increase their functionalities, forming the Internet of Things (IoT). For example, a smart door detects that the last person has left the building and locks itself. If someone is leaving the building, the information can be obtained from their mobile phone. The phone has access to the current location of its user or communicates with the lock via Bluetooth on leaving. The information that no one is left in the building can be used for additional smart functionalities. For example, the temperature can be regulated, or the lights can be turned off to save energy. Access to personal data is needed to make those decisions. In the above example, it is the location. Another example is a smartwatch calling for help in case of an emergency. To make this possible, it measures the heart rate, blood oxygen, and other health indicators.

The collection of sensitive data by the device requires trust by its users. The devices are not always developed securely and can leak personal data. In the past, employees of an

[1] "Number of internet of things (IoT) connected devices worldwide in 2018, 2025 and 2030," https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/, accessed: 17.10.2020

IoT company had access to videos from security cameras sent to their remote servers.[2] Even medical devices had vulnerabilities that allowed attackers to take control. In a worst-case scenario, this can result in the death of patients.[3] Furthermore, a large botnet, Mirai, was responsible for distributed denial of service (DDoS) attacks back in 2016. The botnet was largely built with smart devices and affected more than 300,000 devices.[4]

IoT devices are repeatably in the news with new privacy leaks and vulnerabilities. The reasons for this are diverse. For example, many devices have low resources, which can be problematic when computing complex cryptography. Others are not receiving updates regularly and therefore cannot react to published security problems.

One possible countermeasure is to analyze smart devices for security and privacy issues automatically. Automatic analysis helps developers detect issues before releasing the device. As a result, problems can be fixed before they get exploited. At the moment, this is still an open problem. One of the main reasons is that the hardware and the software of smart devices are diverse. This diversity makes it hard to design general analysis techniques. This thesis aims to develop an analysis technique for detecting personally identifiable information (PII) leaks, which is usable for large-scale analysis of a diverse set of devices.

## 1.2 Motivation

The goal of the thesis is to develop a static analysis approach that provides an overview of how IoT devices are handling sensitive data and is helping to improve the current situation. IoT devices have companion apps to allow users to interact with them. The advantage of analyzing companion apps is that they are developed for mobile platforms such as Android and iOS. Therefore, the programming languages are not as diverse as they are for smart devices. In addition, performing static analysis does not require the hardware of IoT devices. Instead, it analyzes the application statically without executing it. This has the advantage that an analysis can be made with freely available applications from app stores instead of buying physical devices.

[2] "Report Claims Ring Employees Had Unfettered Access To Security Camera Footage," https://www.forbes.com/sites/paullamkin/2019/01/11/report-claims-ring-employees-had-unfettered-access-to-security-camera-footage/#26faea2c206e, accessed: 17.10.2020

[3] "Hacking pacemakers, insulin pumps and patients' vital signs in real time," https://www.csoonline.com/article/3296633/hacking-pacemakers-insulin-pumps-and-patients-vital-signs-in-real-time.html, accessed: 17.10.2020

[4] "Mirai botnet: Three admit creating and running attack tool," https://www.bbc.com/news/technology-42342221, accessed: 17.10.2020

Multiple sub-goals need to be fulfilled first to reach the goal of the thesis:

1. Develop a reliable static analysis approaches for companion apps.
2. Distinguish between local and remote communication.
3. Gather an up-to-date list of companion apps for the analysis.

## 1.3 Methodology

First, a literature research is carried out to gather background information about companion apps and static analysis. Then, we consider additional resources like code repositories and blog posts to understand how companion apps are developed. If there are not enough current open-source companion apps, we will decompile and manually analyze selected companion apps to get a better overview.

Next, we begin with the main part of the thesis and develop our static analysis approach. FlowDroid [5] is a well-known tool for Android. For this thesis, we use it as the starting point. We adapt it to meet the requirements for companion apps. FlowDroid can only analyze Android and Java applications. Therefore, we only consider Android companion apps. In the end, our approach must be capable of:

- tracking data flows between Bluetooth and the network API.
- tracking data flows between the local network and the internet.
- detecting the used communication protocols.

We test our analysis approach with sample applications to assure it is working as expected.

Then, we collect some companion apps from the Google Play Store and carry out a larger analysis.

In the end, we manually analyze the obtained results and evaluate them.

## 1.4 Related Work

### 1.4.1 State of IoT Security

Previous work showed multiple attacks on IoT devices. Ho et al. [19] focused their research upon identifying attack scenarios on smart locks. They showed security problems within key revocation and automatic unlocking features. Both can lead to unauthorized access due to design flaws.

Nowadays, even medical devices are getting connected to the internet. Wood et al. [54] took a closer look at the data transmission of such medical IoT devices. They discovered

a blood pressure monitor that leaks information, like the frequency of measurements despite using encryption. In contrast, there are even devices, like a smart hydration tracker, that send data via unencrypted connections [15].

Alrawi et al. [1] tested and rated 45 IoT devices,[5] based on rating criteria they designed. With the rating, they wanted to make it easier for non tech-savvy users to compare the security and privacy of those IoT devices. For designing the rating criteria, they first carried out a literature review. From the literature, they identified four main components of smart devices: the device, mobile application, cloud endpoint, and communication. Their evaluation showed that if a device performs well in one category, this does not automatically mean it is also performing well in the other categories.

In contrast to this thesis goal, the research mentioned focused only on a small number of devices, and none of them have performed a large-scale analysis.

### 1.4.2 Detection of Personally Identifiable Information Leaks

Detecting leaks of private information in apps is a well-known research area. The approaches can be categorized into static and dynamic analysis. For dynamic analysis, the tested application is executed. During execution, personal information is traced. This approach is more resistant to obfuscation than other methods but also comes with drawbacks. A modification of the system, libraries, or the analyzed application is usually needed. Taintdroid [17] or Uraine [40] are examples for this type of analysis.

Network-based analysis can be seen as a subtype of dynamic analysis. For this approach, network traffic is captured while the app is executed. The traffic gets analyzed if it contains confidential information to find leaks. In the network traffic, privacy leaks can be obfuscated, which makes them harder to detect. In addition, leaks can be missed if the corresponding code is never executed while running the test. Therefore, a high code coverage while performing the test is important. In contrast, network analysis is more robust against false positives. Because if personal data is found in the traffic, it was definitely leaked. Examples for network-based analysis are ReCon [43] or VULPIX [53]. VULPIX makes use of network-based analysis as well as static analysis to combine the advantages of both methods. Also, Chen et al. [11] showed that a combination of static and dynamic analysis help to circumvent the drawbacks of using only one analysis strategy. For analysis of companion apps, dynamic approaches have the drawback that the actual IoT devices are required. Therefore, a large-scale analysis is associated with high expenses.

In contrast to dynamic analysis, static analysis does not execute the application, which leads to many difficulties. One of them is that unwinding potential paths may cause path explosion. Path explosion results in run-time problems or incompleteness if it is stopped after reaching a specified depth. It may also result in false positives when

[5] Scores of the 45 evaluated IoT devices, `https://yourthings.info/scorecards/`, accessed: 15.01.2021

leaks are detected in branches that are not reachable during execution. On the other hand, large-scale analysis can be made without interacting with the program and it does not need the actual IoT device. Examples for static analysis are FlowDroid [5] and Argus [50] (also known as Amandroid).

Mahmud et al. [30] used static analysis to detect violations to the payment card standard (PCI DSS). First, they used natural language processing for detecting input fields related to credit card information. In addition, they reviewed the standard to find requirements for mobile apps that handle credit card data. Later, they used static taint analysis to track credit card information to sinks, like logs or networking methods. Finally, they were checking if the flows are compliant with the standard. In total, they found 6 out of 358 applications asking for credit card numbers to be non-compliant with the standard.

### 1.4.3 Companion App Analysis

In the last years, companion apps were subjected to research. Mauro et al. [31] analyzed 32 companion apps and found multiple vulnerabilities, including unencrypted traffic, hardcoded keys, and insecure protocols. One app even used the insecure Caesar Cipher for encryption. Chen et al. [12] fuzzed IoT companion apps to find memory corruption vulnerabilities in IoT devices. With their approach, they evaluated 17 devices and were able to find 15 vulnerabilities. With the help of this strategy, they were saving the trouble of getting firmwares from devices. Still, they needed physical devices for their analysis.

Zhou et al. [56] discovered new vulnerabilities by abstracting the cloud, IoT device, and companion app with the help of state machines. As a result, they were able to find invalid state combinations and unexpected state transitions. Those can lead, among other security problems, to device hijacking. Wang et al. [49] followed another approach to detect vulnerabilities through companion apps. They observed that IoT devices reuse and customize components from each other. Consequently, also vulnerabilities are propagated to other devices if they are using common components. Therefore, they developed a similarity analysis for companion apps. With this analysis, they were able to discover 324 potentially vulnerable devices. Since the analysis does not require any hardware, similar to the goal of this thesis, they were able to analyze in total 2,081 companion apps. In comparison to this thesis, their approach can only discover known problems from other companion apps.

Some previous papers also focused on finding privacy leaks with the help of companion apps. Clik et al. [10] developed a static analysis tool called SainT. With the help of SainT, they were able to find in 138 of 230 apps from the Samsung SmartThings market sensitive data flows. Additionally, they published IoTBench, a test suite containing SmartThings applications with data leaks. In contrast, this thesis focuses on Android companion applications, which are more widespread. WearFlow [46] developed by Tileria et al. is a static analysis approach based on FlowDroid [5]. The focus of WearFlow is on detecting PII leaks across companion apps running on smartwatches and the smartphone.

They also released Wear-Bench as a testing suite for analysis techniques focusing on leaks across wearable apps. With their approach, they were able to perform a large-scale analysis and detect various privacy violations.

Companion apps can not only be used for static analysis. Ren et al. [42] installed in total 81 IoT devices in two labs, one in the United Kingdom (UK) and one in the United States (US) for detecting privacy leaks in network traffic. They automated the interaction over the application for devices that are controllable with companion apps. Consequently, they were able to control the devices automatically and repeat the interaction more often. With this setup, they looked at the network traffic produced by the IoT devices. Furthermore, they examined where devices connect to. Remarkable is that European devices tend to communicate less with third-parties. This behavior might be due to the privacy regulations within the European Union (EU).

Because companion apps communicate with smart devices and clouds, they need to "speak" a common protocol. Researchers used that to find vulnerabilities and learn more about protocols used by smart devices. One protocol frequently used is Bluetooth Low Energy (BLE), which works as follows: the device broadcasts packets with universally unique identifiers (UUIDs). The packets are received by nearby devices, where the corresponding companion app can recognize the UUIDs, pair with the smart device, and further exchange data. Because the UUIDs are typically fixed, Zuo et al. [58] were able to use companion apps to gather UUIDs and later fingerprint smart devices with the collected identifiers. As a countermeasure, they presented the use of dynamic UUIDs, which change after each usage. Therefore, the application needs to compute new identifiers after establishing a connection and send them to the device and the cloud as backup.

Cars are also becoming more and more connected. Modern cars use Controller Area Network (CAN bus) commands for control. Unfortunately, only a few commands are standardized. These commands can even differ between different models of the same manufacturer. Since cars also have companion apps for controlling some functionalities, like playing music, the mobile apps send CAN bus commands directly or indirectly over the cloud, according to Wen et al. [52]. To automatically reverse the commands at a large-scale Wen et al. [52] developed CanHunter. CanHunter makes use of static and dynamic code analysis. With the help of their approach, they were able to discover 182,619 unique CAN bus commands.

Mohanty et al. [32] developed a static analysis approach for finding security problems in hybrid mobile apps. A hybrid mobile app executes web code and shows it in an embedded web browser. This strategy can reduce the development overhead since the core code can be used for Android, iOS, and the actual web. With their analysis technique, they were able to identify different security issues within 102 companion apps.

For detecting unauthorized access to IoT cloud APIs Li et al. [29] developed the IoT-ApiScanner framework. The framework uses dynamic and static analysis methods to find cloud API endpoints in companion apps. Their analysis aims to find API endpoints from which they can access the smart device without authorization. With their approach, they found 21 APIs in 5 smart devices which are vulnerable to unauthorized access.

### 1.4.4 Endpoint Analysis

Another research area of our thesis has connections to mobile requests and domain analysis. Since one part of the thesis is to detect local communication, we have to reconstruct the connection endpoints and further evaluate those.

Shen et al. [45] analyzed which domains receive private information from mobile applications. They evaluated the telemetry data from 17.3 million users, collected by a security product. In the end, they showed which domains received the most privacy-related data and from which country this data is transferred to which other one. In contrast to our thesis, they worked with general data and did not specifically investigate companion apps.

A static approach to gather URLs in Android applications was made by Rapoport et al. [37]. They called their approach Stringoid, where they statically simulate string concatenations. Besides analyzing 30,000 applications, they compared the results from 20 apps with the results generated by a dynamic approach. They showed that both approaches find URLs the other does not. They identified dynamic loaded URLs as one of the main reasons why static analysis cannot detect all URLs. Compared to our work on detecting local communication and reconstructing URL values, their approach does not reconstruct URLs constructed with other objects like `okhttp3.HttpUrl`. More importantly, the information is lost with Stringoid, where the URLs are used in the code. That information is needed for our further flow analysis.

Jin et al. [21] developed MobiPurpose, to give users more knowledge of why apps are collecting their data. MobiPurpose uses machine learning to automatically detect what data is sent and categorize why the data is collected. However, their analysis requires dynamically generated data for the classification. Therefore, we cannot use their analysis to classify the URLs obtained from our local communication search.

## 1.5 Structure

The remaining thesis is structured as follows: in Chapter 2 we explain fundamentals needed for the thesis. In Chapter 3 we give an overview of our approach and in Chapter 4 we show the corresponding implementation details. In the following Chapter 5, we present our results gathered and discuss them in Chapter 6. In addition, we show there the limitations of our approach and give a glimpse of our future work. Lastly, we summarize our thesis and conclude it in Chapter 7.

CHAPTER 2

# Background

## 2.1 Internet of Things

Smart devices often consist of three components. Besides the device itself, there is often a cloud backend and a mobile app component. For example, a smart vacuum cleaner is constantly sending out the taken route to the cloud to keep its users updated. In the cloud, this information is stored and can be used for further calculations. The mobile applications provide the collected information to its users, enabling users to take further control decisions. In our vacuum cleaner example, a user could decide to stop the cleaning process or restart it. The control commands are most likely transferred from the phone over the cloud to the device in this scenario.

Compared to mobile phones and personal computers, IoT devices have other needs. They often have few resources, like computation power, memory, or power supply. Device manufacturers might think it is not worth adding more resources to the smart devices because it would make the device more expensive. In the end, the customer might not buy it because of that. Also, there are IoT devices that need to be small and light, like smartwatches. Consequently, there is no space for a large battery. Another property of IoT is that there is more machine-to-machine (M2M) communication than in previous IT technologies, like mobile apps.

To better fit the properties of IoT devices, new communication protocols are arising, such as MQTT, COAP, LoWPAN, BLE, or NFC. New technologies can always lead to new vulnerabilities, especially when they were not designed with security in mind. Jia et al. [20], discovered problems within MQTT. Those problems allow attackers to perform DoS attacks or read data from other users and perform actions on other devices in the worst-case.

### 2.1.1 Companion Apps

As we have seen in the vacuum cleaner example, the companion app is a central component for IoT devices. The example only provided one possible communication scenario between the device and the phone. For the above scenario, the vacuum cleaner first needs to get connected to the internet. For devices without a user interface, one way to do this is to connect through another communication channel to the app, transmit the required Wi-Fi information (e.g., SSID, password), and later communicate over the cloud if the connection was successful. Another reason why companion apps are communicating over the local network is to transfer large amounts of data. For example, if the screen of a mobile phone is shared with a Smart-TV, the data is usually sent over the local network. Other device types, like smartwatches, are often connected over Bluetooth with the mobile app. For synchronization, the mobile app might forward data to the cloud in such cases.

Another essential functionality companion apps often provide are update mechanisms for smart devices. If a user wants to update the device, a companion app can download the firmware and transmit it over the local network, Bluetooth, or another communication channel to the device. Then, on the device, the update is finally executed. From a security perspective, the update mechanism is especially interesting because many things can go wrong. In the worst case, an attacker could take over the device or make it unusable. Another less harmful scenario is that an attacker can download it and reverse engineer it to get further insights into the device.

Since the companion app is a central part of many IoT devices, it can contain lots of information about devices, like communication endpoints, flows of personal data, communication protocols, or information about firmware updates.

## 2.2 Android

Android is an open-source operating system for mobile devices. Since the first Android device was launched back in 2008[1] it massively grew. In 2020, Android had a market share of 72 %.[2] A primary reason for its success are the additional functionalities provided through apps. In the official Google Play Store,[3] are more than 2.8 million apps to download.[4]

Most apps are written in Java or Kotlin, but it is also possible to write code in C for better performance. In the end, apps are getting compiled and distributed as APK files.

[1] "The history of Andorid: The evolution of the biggest mobile OS in the world," https://www.androidauthority.com/history-android-os-name-789433/, accessed: 05.02.2021

[2] "Mobile Operating System Market Share Worldwide," https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed: 05.02.2021

[3] Google Play, https://play.google.com/store/apps, accessed: 05.02.2021

[4] "Number of apps available in leading app stores as of 3rd quarter 2020," https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, accessed: 05.02.2021

```
1 |-- AndroidManifest.xml   (contains information among others about the
    ↪ app version, permission and components)
2 |-- classes.dex           (compiled code in dex format, in the case there
    ↪  are more than 65,536 methods, it needs to be split up into
    ↪ multiple dex files)
3 |-- kotlin/               (additional information for the compiler for
    ↪ Kotlin classes, since the app or some part of the app (libraries)
    ↪ are written in Kotlin)
4 |-- lib/                  (libraries)
5 |-- META-INF/             (manifest file containing the list of files
    ↪ with their signature, certificate information, version information)
6 |-- res/                  (resources not in resources.arsc)
7 |-- resources.arsc        (compiled resources)
```

Listing 2.1: Sample APK content

An APK is basically a ZIP file containing compiled code and other resources. Sample content of an APK file is shown in Listing 2.1.

Every app is running in a sandbox because of security and privacy reasons. In more detail, every app gets a user ID assigned, which is used for restricting the access of files belonging to apps. Additionally, every app is running in an own virtual machine (VM), according to Android [2].

If an app needs access to privacy sensitive data, the app needs to request the corresponding permissions. For example, if an application wants to access Bluetooth, it declares it in the Android manifest file. Before using Bluetooth functionalities, the permission must be requested from the user. If a user does not grant permission, the app usually cannot provide its full functionalities. Not all permissions need to get requested at runtime. Some, like the internet permission, are automatically granted on installation when declared in the manifest. That apps request permission at runtime and users can manage them individually was not always the case. Before Android version 6.0, it was only possible to accept or deny all permissions at installation. If permissions were not granted, it was therefore not possible to use the app. Even if runtime permissions are a step in the right direction, they cannot completely prevent PII leaks. One scenario is that a permission is granted to an app for a legitimate use case, but the app misuses the trust later. For example, a companion app requests permission to scan the local network to find a corresponding IoT device. Afterward, the companion app could continuously scan the network and transmit the results to its servers even if it is not intended from the user anymore. Of course, it is possible to revoke granted permissions, nevertheless the average user hardly ever does it. Reardon et al. [41] showed that apps send out personal data for which they do not have permissions. They found a library saving the unique IMEI number encrypted on the SD card if an app that uses the library has permission to read the IMEI. Later, other apps using the library were also found transmitting the IMEI even if they did not have the corresponding permission. Regarding companion apps,

permissions provide only limited protection. Companion apps might leak data initially received from the smart device, which is not manageable via Android permissions.

Besides declaring permissions inside the Android manifest, the manifest contains other essential information. For example, the targeted Android version is specified there. Usually, also a minimum Android version is set too. In addition, every component of the app needs to be declared in the manifest file. There are four different components in Android:

- **Activities** representing single screens of applications. For example, in a vacuum cleaner robot app, a screen containing a map from the last cleaning can be one activity and a screen to set a schedule for the robot another one.
- **Services** are components running in the background and performing long-running tasks. Back to the vacuum cleaner robot app, a service can be used to constantly receive the current location of the vacuum cleaner in the background.
- **Broadcast receivers** allow apps to listen to broadcast events. Apps can even receive events if they are not running.
- **Content providers** are managing the data of apps. Like saving data in an SQL database or storing a file. If content providers allow it, other apps can use them to read or change data belonging to another app.

## 2.3 Static Analysis

One approach for static analysis is data flow analysis. Therefore, paths personal data can take through the program are computed. Taint analysis can be seen as a form of flow analysis, which can be used to find PII leaks. Taint analysis is similar to debugging. Variables are getting "tainted" if problematic values are assigned. An example from debugging is investigating a null pointer exception. Therefore, null value assignments are searched. Afterward, the value is getting traced through the program. If it reaches a critical statement, a possible problem is reported. For PII leak detection, critical statements are among others those which send data out. For taint analysis, methods returning critical values are called sources, and the statements leaking critical values are called sinks.

We provided an example in Listing 2.2. At line 3 a source `locationManager.getLastKnownLocation(provider);` is called and the variable `lastLocation` gets tainted. At the next line, the method `sendRequest` is called with the tainted variable. As a consequence, the taint is carried further to the parameter of the method. At line 9, the tainted parameter is concatenated, and the result is getting tainted. Since the concatenation result is used for building an object, the whole `request` object is getting tainted (line 8). At line 12, a sink is called with a tainted value. Therefore a taint analysis tool would report a leak.

```
1  public void leakExample() throws IOException {
2      LocationManager locationManager = (LocationManager) getSystemService(
           ↪ LOCATION_SERVICE);
3      Location lastLocation = locationManager.getLastKnownLocation(provider
           ↪ );  //source
4      sendRequest(lastLocation);
5  }
6
7  public Response sendRequest(Location location) throws IOException {
8      Request request = new Request.Builder()
9              .url("http://some-backend.com/" + location.toString())
10             .build();
11
12     return client.newCall(request).execute();   //sink
13 }
```

Listing 2.2: Taint analysis example

Due to the amount of different static analysis approaches for Android, researchers found the need to compare the different techniques [9, 28]. Even if there are many different approaches out there, many of them are based on Flowdroid [5] or Argus [50].

FlowDroid [5] is an open-source static taint analysis tool. Its focus is on Android applications in general, which have several additional challenges compared to Java applications. FlowDroid is based on the Soot framework [48]. Soot was originally made for Java bytecode optimization. Nowadays, it is widely used for static analysis approaches. For tracking data flows, FlowDroid makes use of the IFDS [44] framework. In contrast to the idea of this thesis, FlowDroid is generally built for Android applications and does not satisfy the specific needs for analyzing companion apps out of the box. FlowDroid is still actively developed (latest released version 2.8 from July 2020 and new commits every few days)[5] and overall performs well in comparisons. Therefore, we are using it as a base component for this thesis. Also, many other researchers before extended FlowDroid to better match the needs for their analysis, some examples are [26, 27, 34, 46, 53].

Argus [50] is another analysis program. Although FlowDroid inspired it, it has some fundamental changes. The main difference is that Argus focuses on tracking inter-component communication (ICC), which is only rudimentary integrated into FlowDroid. As for FlowDroid, Argus is generally built for Android applications. Therefore it does not perfectly fit the needs of companion apps out of the box. We did not use Argus for the analysis, even if it supports useful features for the value set analysis. Since it is not actively developed anymore, the last commit is two years old (January 2019).[6]

[5] FlowDroid Github, https://github.com/secure-software-engineering/FlowDroid, accessed: 08.02.2020

[6] Argus-SAF Github, https://github.com/arguslab/Argus-SAF, accessed: 08.02.2020

### 2.3.1 Terminology

In this subsection, we summarize essential terminology frequently used in combination with static analysis.

- **Sound:** If the analysis is sound, every reported leak must be truly a leak. Consequently, a sound analysis does not produce any false positives. Nevertheless, an analysis is still sound if it misses out on leaks. In an extreme case, the analysis is sound if it never reports any leak. Such an analysis would not add any value. Worse, it gives a false feeling of safety.

- **Complete:** A complete analysis does not produce any false negatives. Such an analysis finds all privacy leaks in the program. An analysis is still complete if it reports more privacy leaks than there are. An analysis could report everywhere privacy leaks to be complete. However, such analysis is also worthless.

- **Precision:** Precision is the fraction of true positives from totally reported results (true positives and false positives). It can be computed as follows: $\frac{true\_positive}{(true\_positive+false\_positive)}$ [36].

- **Recall:** Recall is the fraction of the found leaks from the totally existing leaks (true positive + false negative). In practice, it is extremely hard to calculate the recall of analysis. The whole program needs to be analyzed manually to know how many false negatives there are. Even if this is done, leaks can be easily missed, which results in a wrong recall. The recall can be computed as follows: $\frac{true\_positive}{(true\_positive+false\_negative)}$ [36].

- **Flow-Sensitivity:** If an analysis is flow-sensitive, it is aware of the statement order. For example, in Listing 2.3 a flow-insensitive analysis could conclude that data from the humidity sensor and the temperature sensor are sent out. While a flow-sensitive analysis knows that the temperature sensor was assigned after the data was sent out. Flow-insensitive analysis can switch statements in order to simplify the graph generation, for example [28].

- **Field-Sensitivity:** If an analysis is field-sensitive, it keeps track of the combination field and object. Field-insensitive analysis taints the whole object if a single filed of the object gets tainted. In Listing 2.4 a field-insensitive analysis detects two leaks in line 11 and line 12 because the whole object gets tainted in line 10. A field-sensitive approach distinguishes between the different fields and therefore only reports the correct leak in line 11 [3].

- **Context-Sensitivity:** A context-sensitive analysis considers the calling context while analyzing the called function. In Listing 2.5 a context-insensitive analysis is not able to distinguish if the humidity or temperature sensor called `readValue` [28].

```
1  public void flowSensitivityExample(){
2      Sensor sensor = new HumiditySensor();
3      sensor.sendValue();
4      sensor = new TemperatureSensor();
5  }
```

Listing 2.3: Flow-sensitivity example

```
1  public class TemperatureData implements Sensor {
2      String[] symbols = new String[]{"C", "F"};
3      Float temperature = null;
4      int unit = 0;
5      //...
6  }
7
8  public void fieldSensitive() {
9      TemperatureData temperatureData = new TemperatureData();
10     temperatureData.temperature = readSensorData();          //source
11     sendData(temperatureData.temperature);
12     sendData(temperatureData.symbols);
13 }
```

Listing 2.4: Field-sensitivity example

- **Access Path:** An access path is the sequence of fields in order to access a value. In Listing 2.4 the access path in line 11 is `temperatureData.temperature`, for example. One thing to consider while working with access paths is that there could exist infinity long ones. For instance, in recursively defined data structures. Static analysis tools limit the length and truncate everything afterward to handle this problem. For example, if the limit is 2, `data.prev.prev.prev` is handled as `data.` ↪ `prev.prev.*`. As a consequence, the analysis is losing precision but is at least able to handle it [3].
- **Program slicing:** Program slicing was introduced by Weiser [51] as a debugging strategy. Program slicing intends to reduce the complexity regarding a specific property without changing the interesting program behavior. For example, if the task is to compute possible values of a URL at a specific point in the program, it can be helpful to extract the corresponding program slice. This slice needs to contain all operations involved in the URL computation.

```
1  public Sensor getSensor(Sensor sensor){
2      return sensor;
3  }
4
5  public void contextSensitivityExample() {
6      Sensor sensor;
7      Sensor humidity = new HumiditySensor();
8      Sensor temperature = new TemperatureSensor();
9
10     sensor = getSensor(humidity);
11     sensor = getSensor(temperature);
12     sensor.readValue();
13 }
```

Listing 2.5: Context-sensitivity example

### 2.3.2 Challenges

As well known, the halting problem is undecidable. Consequently, a perfect static analysis with no false positives and false negatives is not possible. Moreover, even single sub parts of static analysis are undecidable, as shown by Landi [24].

As a result, developers of static analysis have to make trade offs between completeness and soundness. Bonett et al. [9] took a closer look at design decisions regarding soundness and completeness of static analysis approaches for Android apps. For future developers, the fact that not all those decisions are well documented can turn out as a problem. If another analysis is built on top of the original, the problem also gets inherited. For detecting unsound design decisions, the researchers build a framework and reported previously undocumented ones.

#### Android Specific Challenges

In addition to the general challenges of static analysis, there are several Android specific ones. Some of them are inherited from Java, as: reflection, native code or multi-threading others are not.

The first Android specific challenge is converting apps byte code into an intermediate representation (IR) for the analysis. Static analysis approaches usually work on an IR to make further computations easier. FlowDroid uses Dexpler [7] for converting Dalvik byte code of apps into the Jimple format the analysis is using.

In comparison to most other programming languages, Android applications do not have a main function. Instead, there are several different program entry points. Figure 2.1 shows the lifecycle of activities. As we can see, the activity is entered over the `onCreate()` method if the activity is opened the first time. Afterward, the user might switch to another app and later continue using the first app again. Then the component is entered over the `onResume()` method instead of the `onCreate()` as before.

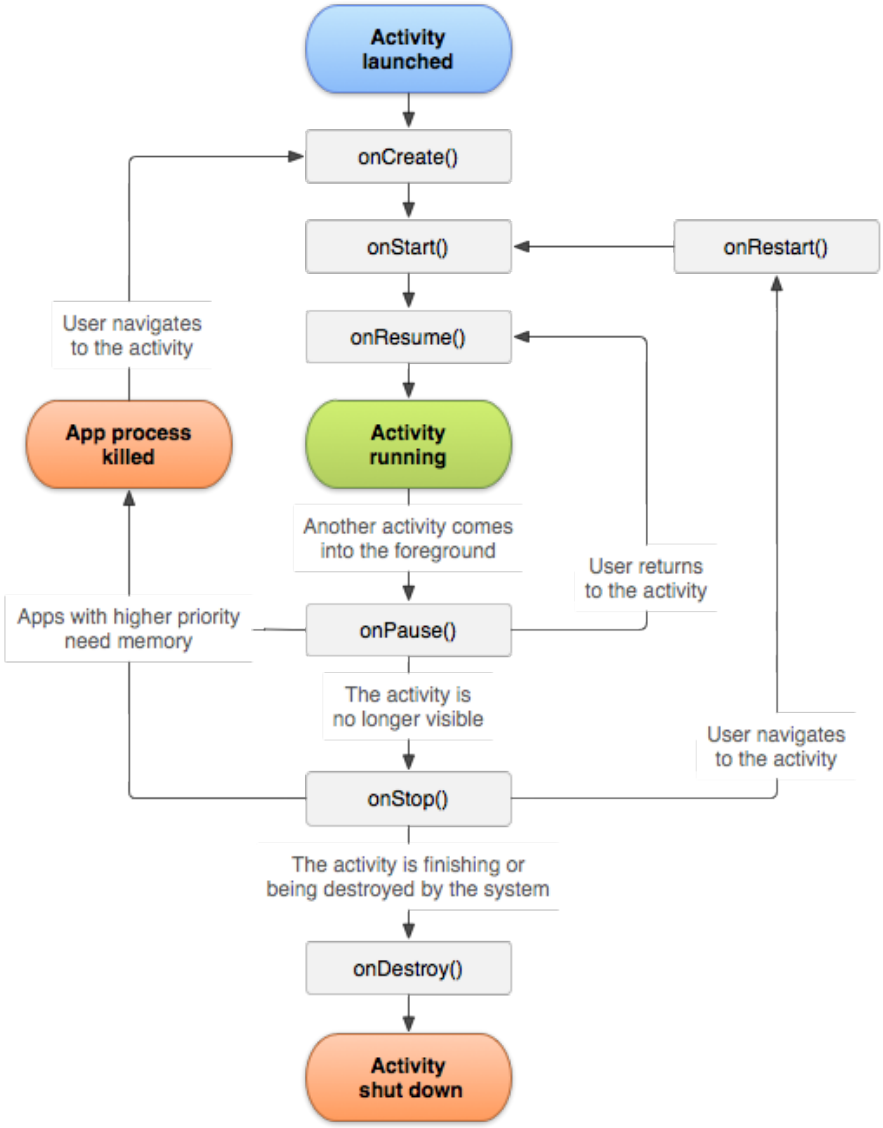


Figure 2.1: Activity lifecycle (Figure from: `https://developer.android.com/guide/components/activities/activity-lifecycle`, accessed: 09.02.2020)

```
1 public void computations(int toAdd, int toMultiply, String baseString) {
2     int i = 1;
3     i = (i + toAdd) * toMultiply;
4     String newString = baseString + "/";
5     String anotherNewString = new StringBuilder().append(baseString).
          ↪ append("/").toString();
6 }
```

Listing 2.6: Java code which was the basis for Listing 2.7

As mentioned in Section 2.2 apps can interact with components from other apps. Also, the app itself can use mechanisms like intents to communicate between its components. This communication is called Inter Component Communication (ICC). However, the ICC makes it hard for analysis techniques to track flows between different components.

While developing Android applications in Java or Kotlin, it is also possible to write parts in C code. Most Android static analysis techniques can handle only rudimentary C code. Therefore leaks could get hidden within the native code. Fortunately, researchers [38] found out that only approximately 14.5 % of apps make use of native code in 2015.

Just like for regular Java programs, it is possible to load code in Android applications dynamically. Since static analysis is not executing the application, it also cannot analyze the dynamically loaded code.

### 2.3.3 FlowDroid

This subsection contains details about different components of FlowDroid and the IFDS framework FlowDroid's analysis is based on. The analysis performed by FlowDroid is *context-sensitive, flow-sensitive, field-sensitive, and object-sensitive*, which are excellent preconditions for static analysis.

#### Jimple

FlowDroid is using intermediate representations for its computations. The default representation used from Soot and FlowDroid is called Jimple [47]. Byte codes like the Dalvik or Java byte code usually have lots of different operations. In comparison, the representations used from static analysis tools are much simpler. While byte code is optimized for execution speed, Jimple is optimized for analysis tools. As a result, there are fewer operations, and instead of a stack, local variables get introduced. In addition, nested instructions are getting split up into simpler ones. All those mechanisms make life easier for static analysis. An example Jimple code is shown in Listing 2.7 with the corresponding Java code in Listing 2.6.

```
1  public void computations(int, int, java.lang.String)
2  {
3      com.example.bluetoothdevice.JimpleExample r0;
4      int $i0, $i1;
5      java.lang.String $r1, $r3;
6      java.lang.StringBuilder r2;
7      r0 := @this: com.example.bluetoothdevice.JimpleExample;
8      $i0 := @parameter0: int;
9      $i1 := @parameter1: int;
10     $r1 := @parameter2: java.lang.String;
11
12     $i0 = 1 + $i0;
13     $i0 = $i0 * $i1;
14
15     r2 = new java.lang.StringBuilder;
16     specialinvoke r2.<java.lang.StringBuilder: void <init>()>();
17     virtualinvoke r2.<java.lang.StringBuilder: java.lang.StringBuilder
           ↪ append(java.lang.String)>($r1);
18     virtualinvoke r2.<java.lang.StringBuilder: java.lang.StringBuilder
           ↪ append(java.lang.String)>("/");
19     $r3 = virtualinvoke r2.<java.lang.StringBuilder: java.lang.String
           ↪ toString()>();
20
21     r2 = new java.lang.StringBuilder;
22     specialinvoke r2.<java.lang.StringBuilder: void <init>()>();
23     virtualinvoke r2.<java.lang.StringBuilder: java.lang.StringBuilder
           ↪ append(java.lang.String)>($r1);
24     virtualinvoke r2.<java.lang.StringBuilder: java.lang.StringBuilder
           ↪ append(java.lang.String)>("/");
25     $r1 = virtualinvoke r2.<java.lang.StringBuilder: java.lang.String
           ↪ toString()>();
26
27     return;
28 }
```

Listing 2.7: Jimple code generated from the byte code of Listing 2.6

### IFDS

IFDS stands for inter-procedural, finite, distributive, subset, all properties a problem needs to have to apply the IFDS framework. Since the IFDS framework was presented [44] there have been improvements [33], among others FlowDroid switched from Soots IFDS solver [8] to a more efficient one [3]. Using the IFDS framework for computing data flows combined with access paths for taint abstractions enables FlowDroid to be context-sensitive, flow-sensitive, and object-sensitive. The general idea of IFDS used in FlowDroid is to generate a graph and compute the reachability of taints on it. More concrete starting from an inter-procedural control flow graph, FlowDroid creates a so called exploded supergraph. That means each node is also holding facts. The fact 0 is a special fact which always holds. For the flow analysis, the facts represent tainted data. The exploded supergraph contains an edge between two nodes n1, n2 if there is a connection in the Interprocedural Control-Flow Graph (ICFG) from n1 to n2, and the fact holding at n2 also previously held in n1. Flow functions are used to compute the facts for the successor nodes. The flow function can generate, kill, or retain facts/taints. Depending on the current statement, FlowDroid distinguishes between the following four types of flow functions:

- **Call flow functions** handle method calls. They are applied on the call site and handle the mapping from the arguments to the parameter. In addition, call flow functions need to take care of the base object from a method if there is one.
- **Return flow functions** get triggered if a method is left. That can happen either due to a return statement or when an uncaught exception is thrown. The return flow function does the opposite of the call flow function. Therefore, it has to map the parameters back.
- **Call-to return flow functions** are method calls which skip the callee. This rule is used for excluded methods, which get treated as black boxes, for example.
- **Normal flow functions** are statements not handled by call nor return flow functions like assignments, arithmetic computation, or conditions.

Based on the flow function, the decisions to generate, kill or retain the taints are taken. In Figure 2.2 an example from FlowDroid [3] is provided. In the first line `a = source();` the variable `a` is getting tainted. The statement is handled from the call to return flow function rule because we treat the function as a black box that returns a tainted value. This means that a new taint is generated for variable `a`, symbolized with an edge from the special fact `0` to the variable `a`. The variables `c.a` and `b` are not involved in the computation. Therefore, their facts are retained. The second program statement `b = a;` gets handled by a normal flow function. The fact of variable `a` gets passed on to variable `b` symbolized in the graph by the edge. Also, the other values are not modified. Therefore, they have edges from their predecessor. Next, a call flow needs to be handled at the statement `callee(b, c);`. The variables `b` and `c` are used as parameter for the function

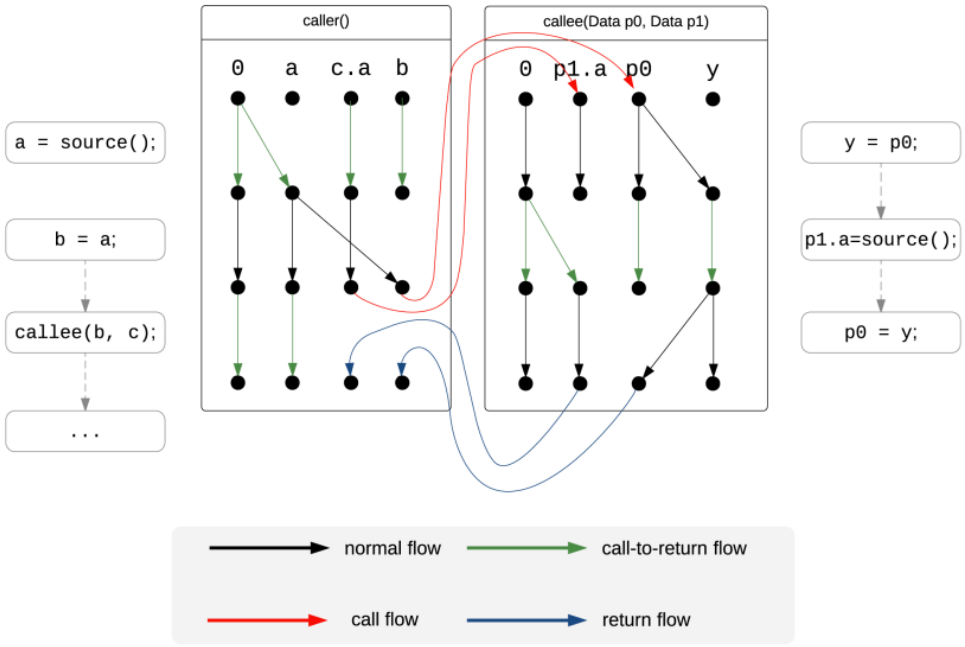


**Figure 4:** Four Types of IFDS Flow Functions

Figure 2.2: IFDS example from FlowDroid [3]

callee. As a result, there are edges from `b` and `c.a` to the local variable of the function callee `p1.a` and `p0`. The next three statements inside the callee functions are again normal flows and a call-to-return flow as we had it before. After the last statement of the callee function, the control flow returns back into the caller. The taints need to be mapped back. In the figure, the return flow is symbolized with the blue arrows.

### Alias Analysis

Listing 2.8 shows an example where alias analysis is needed. Otherwise, it would miss the leak in line 10 (assuming `getCurrentLocation` is a source and `sendOutData` a sink). Static analysis needs to search for aliases to circumvent missing a leak. The search is done in the opposite direction then the analysis is performed. Therefore, FlowDroid performs it backwards. There are two different strategies when alias analysis is performed: either if a new variable is tainted or if a heap variable is read. Both strategies have their disadvantages. Performing the alias analysis when a variable is tainted might result in more unnecessary alias searches. In addition, if aliases are found, it might be the case that they are carried unnecessarily onwards. On the other hand, performing the alias analysis when a heap variable is read can be very complex and computation expensive in

```
1  public class AliasExample {
2      public class SomeDTO {
3          public Location location;
4      }
5
6      public void aliaExample() {
7          SomeDTO currentLocation = new SomeDTO();
8          SomeDTO locationCopy = currentLocation;
9          currentLocation.location = getCurrentLocation();
10         sendOutData(locationCopy.location);
11     }
12 }
```

Listing 2.8: Alias analysis example code

some edge cases. Nevertheless, normally the lazy strategy is used.

FlowDroid uses the IFDS framework for the alias analysis. To achieve this, the CFG is getting inverted, and a new IFDS solver instance is started. If an alias is found, it is getting injected into the normal forward solver, handled, and propagated as a new taint.

### Taint Propagation Wrapper

To make FlowDroid practical, it is not analyzing the internals from Java and Android libraries. Otherwise, it would take more time to analyze the libraries than the app. In the worst case, the analysis would need much more resources. Therefore the analysis would not finish within a reasonable time. The method calls to excluded libraries still need to be simulated to circumvent losing precision. In order to achieve this, FlowDroid works with pre-defined rules provided by the `EasyTaintWrapperSource.txt`. The rules provide information about taint handling, if new taints are getting generated, killed, or all existing taints maintained because the handled method is excluded. Excluded methods are not changing existing taints. As an example, the rules from Listing 2.9 have the following meaning:

- The first line containing the method `addAll`, defines that the *Set* is getting tainted if an already tainted collection is added.
- Since the `clear` method removes all objects from the *Set*. Also, the taint needs to be killed if the *Set* was tainted before.
- The last line does not generate new taints and not killing old ones. Therefore, it is excluded and returns the taints as they are.

Every handled method needs to be added to the wrapper list manually. For many use cases, it is helpful to exclude widely used third-party libraries, like networking or

```
1 <java.util.Set: boolean addAll(java.util.Collection)>
2 -<java.util.Set: void clear()>
3 ~<java.util.Set: int size()>
```

Listing 2.9: Sample taint propagation rules for Java Set methods

advertisement libraries. The advantage of excluding library code is, as before, the speed up of the analysis. To address this problem Arzt and Bodden developed StubDroid [4]. StubDroid can compute taint rules for library code, which are saved in XML files. FlowDroid can later reuse those rules for the taint analysis. Since the library code is not analyzed repeatedly for every app, the overall procedure is sped up.

### 2.3.4 Value Set Analysis

This subsection gives an overview of value set analysis and shows approaches using this analysis

Static analysis is not only used for detecting privacy leaks. Another use case is extracting possible values at specific points in programs. As an example, for analyzing if data is sent to smart devices or remote servers, it is interesting to know the endpoints called from the program. One possibility is to extract this information by computing values of Uniform Resource Locator (URL) passed to network methods by the program. The reason why this is no trivial problem is that URLs are not always constant values. Indeed, they often get constructed during program execution. Consequently, the URL construction needs to get traced through the program and the steps taken simulated.

Gadient et al. [18] build a static analysis approach upon the *JADX* decompiler. In addition to the URL values, they also reconstructed JSON values. With the data obtained from their analysis, they searched for potential security problems, like programming code in web APIs or disclosure of server version information. Compared to approaches building upon Soot, they do not have the advantage of an IR made for static analysis. Consequently, their approach built upon the source code has to handle more instructions. As an example, Gadient et al. mentioned they have to handle `StringBuilder.append()` and "+" operation for string concatenation. In an analysis based on Soot only `StringBuilder.append()` needs to be handled since Soot is not decompiling the code back into its source code. Also, the other advantages from the specific guarantees of the Jimple IR are not available when performing the analysis on the source code or decompiled code.

#### LeakScope

Zuo et al. [57] identified vulnerabilities of cloud application programming interface (API) keys in mobile applications, e.g., using a root API key inside the mobile application or wrongly configured permissions of a key. To test if an app has such a vulnerability, they needed to extract the keys first. For this task, they built a static analysis approach

```
{
  "apk":"example/ValueSetAnalysisExample.apk",
  "methods":
    [
      {
        "method":"<com.microsoft.azure.storage.CloudStorageAccount: com.
            ↪ microsoft.azure.storage.CloudStorageAccount parse(java.lang
            ↪ .String)>",
        "parmIndexs":[0]
      }
    ]
}
```

Listing 2.10: Example LeakScope config from: `https://github.com/OSUSecLab/LeakScope`, accessed 17.02.2021

called LeakScope, which makes use of value set analysis [6]. In the end, they performed a large-scale analysis and were able to find 17,299 vulnerabilities. For another paper, Zhao et al. [55] used LeakScope to identify hidden behaviors such as master passwords, backdoors, or "Easter eggs."

To reconstruct possible values, LeakScope transforms the binary code into Jimple and creates a CFG with Soot. LeakScope builds a data dependency graph (DDG) based on the CFG. In a DDG, the program instructions are represented as vertices, and if there are def-use dependencies between two vertices, edges are added. After the creation of the DDG, backward slicing is applied. Therefore, methods interesting for the analysis (point of interest) are searched within the application. Those methods are predefined and passed to LeakScope at the start of an analysis. Listing 2.10 shows a configuration for reconstructing the first parameter of `CloudStorageAccount parse`. If LeakScope finds usages of those methods, it collects all computations involved to build the first parameter in a program slice. Afterward, the execution of the program slice is simulated, which results in possible values for the variable at the point backtracking started. Since LeakScope was developed for extracting API keys, it is only tracing and reconstructing string values.

### Extractocol

With Extractocol [13, 22] it is possible to extract information about HTTP transactions. It reconstructs information about requests, as URIs, headers, or request bodies statically. In addition, it is extracting data about responses, like JSON keys. For extracting that information following approach is used:

1. First, so called "demarcation points" are searched. Demarcation points are program statements that split the request and response. For example `Response response = client.newCall(request).execute()` takes a request object and returns a

response object. Therefore, it is a demarcation point. From this point, backward taint propagation is used for tracing the request parameters and forward taint propagation to reveal the data dependences of response objects. The forward, as well as backward traced dependences, are called program slices. A slice encapsulates the corresponding information for a request or response and therefore abstracts irrelevant information away.

2. The second step performed by Extractocol is the signature extraction. Therefore, the program further splits the obtained slices from the first step into the URI, request body, and response body part. Next, the signatures get extracted. That means that Extractocol is simulating the operations and therefore computing the objects used within the network requests. To be able to extract interesting signatures, Extractocol model popular networking, JSON, and XML libraries.

3. As the last step, Extractocol is performing taint analysis to identify dependencies between different slices. With this strategy, Extractocol can find response objects that are later used for constructing a request.

Despite some similarities to our strategy for extracting URI information, we have not built upon Extractocol for several reasons. First, Extractocol is not actively developed anymore. Second, since Extractocol is directly modifying FlowDroid and little documentation is available, it is hard to build upon. Besides, as the authors mentioned in their paper, it is not built for large-scale analysis, which contradicts the idea of our work.

# CHAPTER 3

# Methodology

Based on the literature and observations made while analyzing companion apps manually, we found two aspects of companion apps we focused our flow analysis on:

1. The initial connection process of the smart device with the mobile phone is often done over the local network.
2. Companion apps often use different protocols to communicate than other apps.

## 3.1 Local Communication

We saw that smart devices often communicate over a local network to the mobile phone during the pairing process. If smart devices can connect to wireless networks and do not have a graphical user interface on their own, a standard procedure therefore is:

1. The smart device spawns an own personal network allowing the mobile phone to connect. On the left-hand side of Figure 3.1 a corresponding screen from a vacuum cleaner robot is shown.
2. After the mobile phone connects to the network of the smart device, the communication between the companion app and the physical device begins.
3. The user enters on their phone information about their home wireless network. Figure 3.1 shows this step on the right-hand side. As before, the screenshot is taken from a vacuum cleaner robot app.
4. The companion app transfers the information about the access point to the smart device, and the smart device connects to the network.

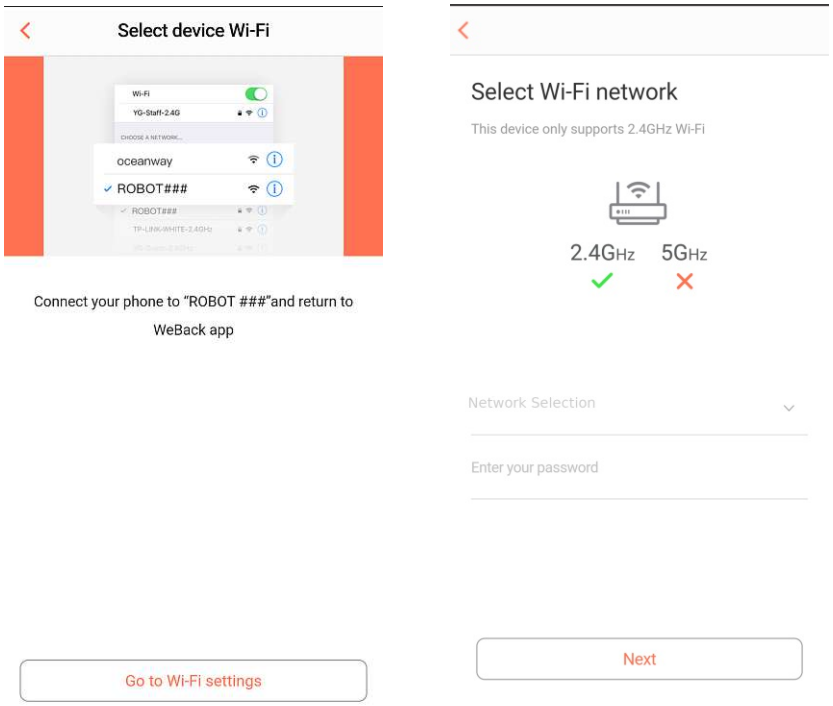


Figure 3.1: The pairing process for a Tesvor vacuum cleaner over the WeBack app `https://play.google.com/store/apps/details?id=com.yugong.Backome`

5. If the setup was successful, the smart device and companion app communicate further over the cloud.

Since the local communication described above is not typical for general apps, our focus is on detecting local connections and trace the corresponding data. To find the local connections, we decided to reconstruct the URLs used in the mobile app at the points where the URL is added to the request. In order to do this, we are using value set analysis. To better fit our usage, we decided to extend an existing analysis. Because we cannot tell beforehand which statements are used for the local communication, we have to reconstruct all URLs in the app. Still, this is no disadvantage since the reconstructed URLs contain countless valuable information. To show the improvements from our extension, we later compare the results obtained from our analysis to the original implementation, regarding time spent on the analysis and found results.

Nevertheless, the initial Wi-Fi connection process is not the only case where smart devices and companion apps communicate over a local network. Other devices have a GUI for connecting to the wireless network like a Smart-TV or are connected via an Ethernet cable but still uses the local home network for providing functionalities that require lots of bandwidth. For example, Smart-TVs often have functionalities for screen mirroring. That means a mobile phone can mirror its screen, which is displayed on the TV. For

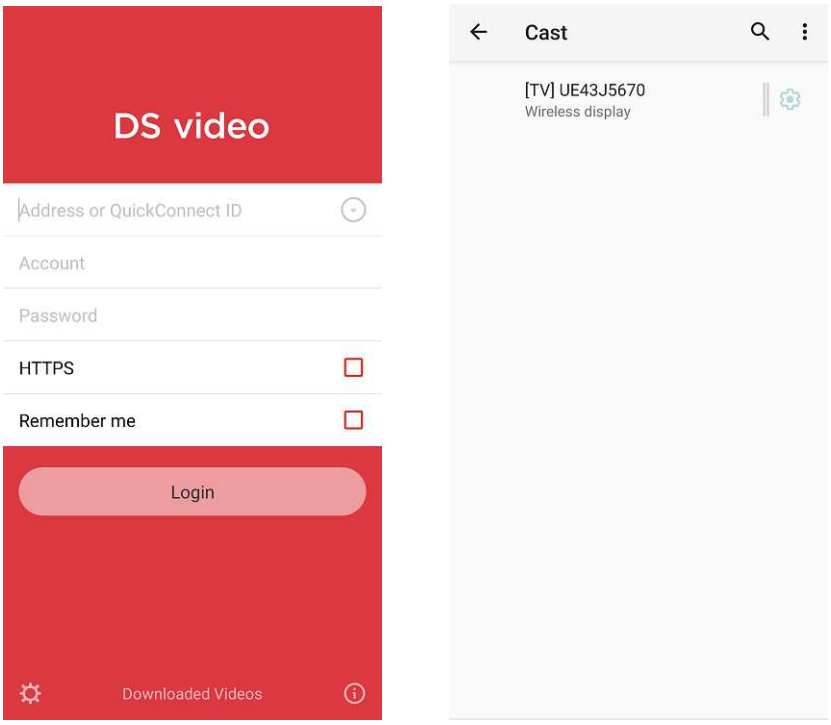


Figure 3.2: Different scenarios for connecting a local device. The left screenshot is taken from the DS video app `https://play.google.com/store/apps/details?id=com.synology.dsvideo`

providing this feature, both devices must be on the same network. The mobile phone will scan the network and ask the users to select the device they want to connect to from the appropriate devices found by the scan. In other scenarios, apps immediately ask the user to enter the local IP address of the IoT device to communicate with them. In Figure 3.2 screenshots are shown, which uses the above explained strategies. To find those local communications, we add default values for user-inputted data to know later which values originate from user input.

For finding the local communication, we recompute URLs that might get passed to predefined network functions. If we find a local IP address or a default value for an IP address symbolizing that it originates from user input, we assume that it is used for local communication.

## 3.2 Endpoint Evaluation

While working on the value set analysis, we came up with multiple interesting questions we wanted to investigate further:

- To which countries are the companion apps connecting?

- Are companion apps connecting more to Asian countries since many IoT device manufacturers are from Asia?
- Are there differences detectable between general-purpose apps and companion apps from the third-party services they call?
- How is the local communication implemented?

To answer those questions, we analyzed selected companion apps and general apps.

Since the collected results by our analysis are not obtained by capturing the network traffic, we cannot evaluate the frequencies of data sent out to IP addresses nor all servers the app connects to. Nevertheless, we know some of the endpoints used. For our evaluations, we extract the domains and IP addresses from the reconstructed URLs. If we extract a domain or IP address multiple times within an app, we only count those once. Otherwise, backends with more paths or URLs with more query parameters would be overrepresented in our evaluations, even if the app might contact them less often. Based on this preprocessing, we further generated graphs and diagrams to answer the questions above.

## 3.3 Flow Analysis

Coming to the second focus, the different protocols used within companion apps. The focus of protocols used by smart devices is on lightweight and energy-efficient protocols due to the limited resource of the devices. We found several sources and sinks for tracing values transmitted over protocols like Bluetooth, BLE, MQTT, or NFC. However, some protocols like ZigBee require additional hardware, usually not built-in today's mobile phones. Therefore, we cannot trace data flows from or to devices over those protocols directly. In such cases, mobile apps often communicate indirectly with the physical device over common protocols. For example, to control the IKEA smart lights[1] a gateway is needed which is connected to the local network. Since we already reconstruct the URLs used and are tracing network methods, we do not have to treat those specially. Besides, the reconstructed URLs can tell us more about the protocols used, for example, if a connection is enhanced with TLS or not.

In the end, we integrate the URL reconstruction for finding local network communication into FlowDroids flow analysis. Besides, we add additional sources and sinks to trace further protocols used by companion apps. With the extended flow analysis, we are analyzing all previously selected companion apps and present our findings.

[1] IKEA Trådfri, `https://www.ikea.com/at/de/customer-service/product-support/smart-lighting/home-smart-beleuchtung-pub1ebae7ef`, accessed: 11.04.2021

CHAPTER 4

# Design and Implementation

In the following sections, we provide more details about our implementations and the extensions to the existing analysis approaches of FlowDroid and LeakScope.

## 4.1 ValueScope

To find local communication, getting insights about the used protocols, knowledge about the corresponding cloud, and the smart device, we decided to extend LeakScope. LeakScope was already used in recent times for various papers [52, 55, 57, 58]. One advantage over other approaches like Extractocol is that it is not as complex and faster since it only has a subset of tasks to perform. It is also quite new (last release from May 2019)[1] in comparison to other approaches [14, 25].

We had to extend LeakScope since it is not designed for reconstructing URLs. As mentioned earlier, it is only capable of tracing string values. However, network functions are not always called with string value arguments. For example, we saw that developers are using the `okhttp3.HttpUrl` which can later be passed to the request builder (`<okhttp3 ↪ .Request$Builder: okhttp3.Request$Builder url(okhttp3.HttpUrl)>`) to construct an executable request. The `okhttp3.HttpUrl` itself can be constructed over another builder. It would still be possible to implement the method simulation only with strings, but we would need to reimplement many different methods and classes. Reimplementing those is not feasible for each class used in combination with network functions. Therefore, we extended LeakScope to reconstruct different types of objects. We are now able to use the original code in the simulation phase instead of reimplementing it. We call our extension ValueScope.

[1] LeakScope Github, `https://github.com/OSUSecLab/LeakScope/releases`, accessed: 16.04.2021

We first give an overview of the extensions we made and then explain the general approach of ValueScope in more detail.

First, we faced several problems concerning dependencies used by LeakScope. To solve this issue, we are now using Gradle[2] as build tool. We also removed several files from the original code since they were just copies from a library and added the library as a dependency to Gradle. Similarly, we cleaned up the whole source code. Where possible, we switched to existing libraries. Besides, we removed unused and unnecessary code. We fixed several bugs found during the whole development process regarding circle detection, static initialization, and missed code paths. However, our main changes to LeakScope were in order to trace arbitrary types of objects and not just strings. Consequently, we had to change the forward computation and the heap object handling. The changes allow us to rely on existing library implementation without reimplementing them to work with strings only. The only downside of these changes is that they add more type casts that are error prone and require extensive testing on changes. At the moment we are modeling the following object types: *Integer*, *Strings*, *InetAddress*, *InetSocketAddress*, *URI*, *URL* and *okhttp3.HttpUrl*.

With our changes to trace arbitrary objects, we also added array handling. Internally we simulate arrays with array lists due to their easier handling. If a value gets written at some array index, we first try to look up the index value. If the analysis is not aware of the index, we append the assigned value to the end of the array. Nevertheless, this can lead to incorrect results. For example, if some values are not in the correct position and the array is used later for a string format, the resulting string is mixed up. Handling unknown read indexes is easier. In this case, all values found for the array so far are returned.

To avoid running into an endless loop LeakScope is not analyzing cyclic blocks. Originally LeakScope was removing all cyclic blocks from the analysis. In the end, this had the effect that values were missed. To avoid missing those values completely, we now check if a cyclic block has already been visited and only remove it from the analysis if it has been visited before.

Additionally, we extended the time watcher. Now it is possible to specify separate timeouts for the backtracking and the forward computation phase. If the time is up, the analysis gets notified. Then the analysis has some additional time to finish and report the already found results.

Figure 4.1 shows the different steps performed by ValueScope to analyze an app. First, Soot gets initialized. It is possible to exclude packages from the analysis to speed up the whole analysis. Among the excluded packages are the Java and Android core package, since we model the relevant functions precisely and are not interested in the internal statements, e.g., how Java internally appends a string. Afterward, all code statements which are not excluded get loaded, and a call graph is built.

[2]Gradle Build Tool, `https://gradle.org/`, accessed: 16.04.2021

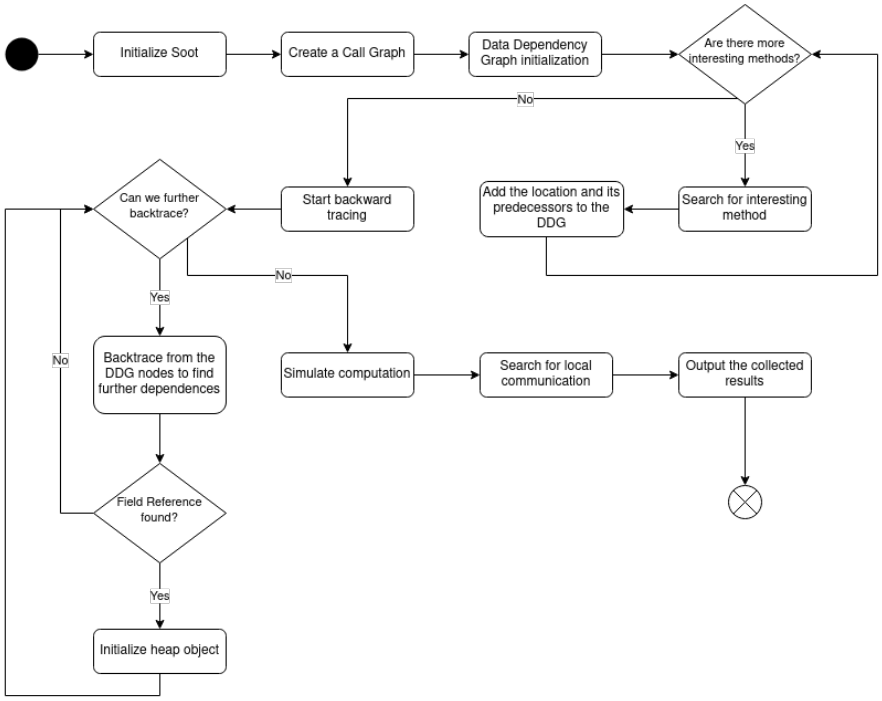


Figure 4.1: Flowchart of ValueScope

As the next step, a data dependency graph is created. Therefore, the method signatures which we are interested in are searched. If there is a usage found, the statement is added to the DDG. This procedure is carried on until all signatures we provided are processed. If there has been a signature found, the backtracking begins. In this step, we keep track of which variables are interesting for the analysis. An own program slice handles each found statement. The program slice has to keep track of its interesting variables. In the beginning, the parameters we are looking for are the only interesting variables. While we keep iterative querying the predecessors, we look for variables and values involved in building those variables. For processing the statements, we have modeled selected methods. For example, if we have the following statement `$r3.<java.lang.StringBuilder: java ↪ .lang.StringBuilder append(java.lang.String)>($r1)` and `$r3` is interesting to us, we add `$r1` to the interesting variables, since it is involved in constructing `$r3`. If the variable of interest is `$r1` and we come across the previous statement, we keep `$r1` in the interesting variable list, but not add `$r3` since this statement is not changing the value of `$r1`. A variable is removed from our list of interesting ones if a constant value is assigned to the variable. An example statement is `$r3 = null`. Since we have traced the variable back to the latest assigned value, we can remove it from our list.

Heap values need special treatment. If a heap object is traced during the analysis, we need to trace back the code which sets a value to the heap object. The backtracking ends either if there is no variable of interest left, we have reached predefined maximum steps

```
1  //..
2  String base = "https://domin.com";
3  String path;
4  if (base.contains("important")) {
5      path = "destination1";
6  } else {
7      path = "destination2";
8  }
9  String urlString = String.format("%s/%s", base, path);
10
11 URL url = new URL(urlString);
12 //...
```

Listing 4.1: Example forward execution

back, or a timeout has been triggered.

Afterward, the analysis progresses with simulating the execution steps involved in building the interesting variables. Therefore, we keep track of possible values for each variable. We provided an example in Listing 4.1. The backtracking will trace the string used to create the URL object in line 11 back to the initialization of the `base` variable in line 2. The forward simulation starts there and saves for the first statement the information that the value of `base` is *https://domain.com*. The next statement declares the variable `path`. Therefore, we save the empty string for this variable. Next, we have an `if` condition. Since we are not evaluating the `if` statements, we do not know that the `else` branch is always entered. To avoid missing possible values, we handle both branches as separate execution. Consequently, we get multiple possible values for the URL in line 11. Lets continue with the example in line 5 we update the value of the `path` variable to *destination1*. In line 7, we need to add another possible value for `path`. To model alternative values, we are saving the possible values in a `Set`. Hence to be more precise, in the above example, we are not saving *https://domain.com* for the variable `base` in line 2. Instead, we create a `Set` and add the string value to it. With this data structure it is now easy to handle line 7 we just add *destination2* into the `Set` from `path`. Since we now have two possible values for the `path` variable, we also need to evaluate the format string twice and add both results into the possible value `Set` from the `urlString` variable. In the end, we get both values as results reported for the URL object in line 11. To simulate an instruction, ValueScope needs to maintain a list of signatures from handled methods. Therefore, the analysis has to look up each parameter and base object for each matching signature if their values are already computed. Otherwise, it takes default values for the simulation. The advantage of taking default values is that partly reconstructed URLs are reported. Those URLs already provide the desired information in many cases. Furthermore, we can use library implementations for the simulation and do not have to implement our own version of the library code because we changed LeakScope to save the variable values as an arbitrary object.

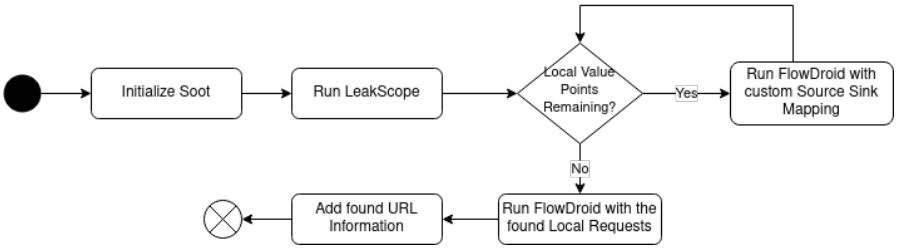


Figure 4.2: Flowchart of IoTFlow

After the simulation finished, we match the results with regular expressions to find local IP addresses. We also add each domain that originated from a UI element as a potential local communication.

At last, the obtained results are saved into a JSON file if requested.

## 4.2 IoTFlow

Our work aims to perform a flow analysis with the additional functionalities as described in Chapter 3. Figure 4.2 shows an overview of the internals of our analysis approach called IoTFlow.

First, we are initializing Soot. Again, we exclude specific packages that are either handled by precise rules or are unlikely to occur in data flows. We then run ValueScope as described earlier, without the Soot initialization since we have already done it.

If ValueScope finds statements that might get used for communicating over the local network, we still do not have enough information for our data flow analysis. In Listing 4.2 we provided an example, if the URL is added in line 2 and we have `<okhttp3.Call: okhttp3.Response execute()>` in our list of sinks, we would only get a data flow reported from line 2 to line 4. However, such data flow does not provide more information than the actual information that in line 2 a local request gets built. Therefore, we cluster the statements found by ValueScope depending on the network libraries or methods used. Our goal is to find the code that performs the network communication from the statement where the local URL was added. To achieve that, we run the flow analysis with a custom source and sink manager. The source and sink manager has the task of answering whether a specific statement is a source or sink. In this step, our custom source and sink manager only returns a statement as a source if ValueScope reported it as used for local communication. The manager reports a statement as sink if the signature matches one of the provided source-sink mapping file. The mappings from sources to sinks are defined initially by the user, a simplified version for the *okhttp* library is provided in Listing 4.3. The mappings have the following meaning if a local communication from one of the three provided signatures is found, the corresponding sinks are the ones at the index 2 of the sinks list below. In this case `execute()` and `enqueue(okhttp3.Callback)` are the corresponding sinks. Since FlowDroid does not keep track of the propagation

```
1            //...
2            Request request = new Request.Builder()
3                    .url("http://192.168.0.1/getData")
4                    .build();
5            Response response = client.newCall(request).execute();
6            String data  = response.body().string();
7
8            //...
9
10           RequestBody postBody = new FormBody.Builder()
11                   .add("leak", data)
12                   .build();
13
14           request = new Request.Builder()
15                   .post(postBody)
16                   .url("https://www.sink.com/")
17                   .build();
18
19           client.newCall(request).execute();
20           //...
```

Listing 4.2: Sample local communication

path of a taint neither of the source statement, the local communication source finding task is performed for each mapping category separately. The flow analysis does not keep track of the taint paths because it merges two taints even if they originate from different sources. In order to provide the tainted paths to the user, FlowDroid recomputes it in the end. Nevertheless, this is an optimization since in most cases less taints need to be propagated.

For the next step, all statements found executing local network requests are added as sources. As previously, the sources and sink manager mark the statements as sources if they were found by the local source finding task. The other sources and sinks are loaded from a provided list.

FlowDroid is already shipped with a predefined list of sources and sinks, automatically generated by SuSi [39]. However, we found several methods not included in the list that make sense to add for companion applications. As bases, we are using the sources and sinks from WearFlow[46]. There are already sources and sinks added which are used for communicating between Android apps and smartwatch apps running on Googles Wear OS.[3] Since companion apps sometimes scan the local network for IoT devices, we added Android's built-in network scanning functions as sources to the list. We also found multiple apps using the MQTT protocol and therefore added methods from

[3]Wear OS, `https://wearos.google.com/`, accessed: 11.04.2021

```
{
    "mappings": [
        ...
      {
        "<okhttp3.Request$Builder: okhttp3.Request$Builder url(java.lang.
            ↪ String)>": 2
      },
      {
        "<okhttp3.Request$Builder: okhttp3.Request$Builder url(okhttp3.
            ↪ HttpUrl)>": 2
      },
      {
        "<okhttp3.Request$Builder: okhttp3.Request$Builder url(java.net.
            ↪ URL)>": 2
      }
      ...
    ],
    "sinks": [
      [
        ...
      ],
      [
        ...
      ],
      [
        "<okhttp3.Call: okhttp3.Response execute()>",
        "<okhttp3.Call: void enqueue(okhttp3.Callback)>"
      ]
    ]
  }
```

Listing 4.3: Mapping example for the okhttp library

the widely used paho library.[4] Besides, we added methods from different networking libraries and various methods used in combination with NFC, BLE, and BL, which were previously missing. A list of sources and sinks we added for our analysis can be found in APPENDIX A.1.

At the last step, IoTFlow is outputting the found data flows. To add further information, we add the information about the endpoints of network requests previously collected by ValueScope. The information can help people analyzing the flow. For example, it makes a difference if personal data is transferred over a connection using TLS or not.

[4]Eclipse paho library, https://www.eclipse.org/paho/index.php?page=clients/java/index.php, accessed: 11.04.2021

## 4.3 Endpoint Evaluation

For further evaluating the data obtained by ValueScope, we developed several Python scripts.

For extracting the domains from the results obtained with ValueScope, we used regular expressions. To either get the domain or subdomain, depending on the analysis, we used the tldextract library.[5] For resolving the IP address either to use it later for IP whois queries, or just having a look if the previously obtained domain seems to be valid, we used the following line of code `socket.gethostbyname(domain)`. For querying the IP location, we used an IP-Whois library[6] for Python.

To generate a world map the GeoPandas[7] library was used. Since the whois library is returning country codes with two letters and the world map uses the three-lettered ones, we first needed to convert them with another library.[8]

For the other graphs and diagrams, we used the following libraries: matplotlib,[9] pyvis,[10] seaborn[11] and wordcloud.[12]

[5] tldextract library, `https://github.com/john-kurkowski/tldextract`, accessed: 04.05.2021
[6] ipwhois library, `https://github.com/secynic/ipwhois`, accessed: 17.04.2021
[7] GeoPandas library, `https://geopandas.org/`, accessed: 17.04.2021
[8] pycountry library, `https://github.com/flyingcircusio/pycountry`, accessed: 17.04.2021
[9] matplotlib library, `https://matplotlib.org/`, accessed: 04.05.2021
[10] Pyvis library, `https://github.com/WestHealth/pyvis`, accessed: 04.05.2021
[11] seaborn library, `https://seaborn.pydata.org/`, accessed: 04.05.2021
[12] word_cloud library, `https://github.com/amueller/word_cloud`, accessed: 04.05.2021

CHAPTER 5

# Results

The results we present in this chapter were produced on a device with an Intel i7-8565U CPU and 16 GB 2667 MHz RAM. We executed the Java Virtual Machine (JVM) with the following additional parameter `-Xmx12800M` to increase the maximum memory size available to the JVM.

## 5.1 Selecting Apps

Kumar et al. [23] provide statistics about the used IoT device types and popular device manufacturer within the different geographical areas. One device type with significant regional differences are surveillance devices. They detected that 54.5 % of IoT devices in South Asia are for surveillance. In comparison, those devices only make up 3.7 % of the total IoT devices in North America. In contrast, home automation devices mainly occur in Europe and North America. Kumar et al. also showed that different regions favor other device manufacturers. One example they provided is Sagemcom, a French company. The devices found by that manufacturer are nearly entirely located in Europe.

With this knowledge in mind, we wanted to perform our analysis on apps belonging to devices used worldwide. Since Kumar et al. provided a list of popular IoT device manufacturers per region and per type, we used that information for our selection. We later searched the Google Play Store for apps from those manufacturers. We selected 130 apps where most of them are belonging to one of those manufacturers. Due to regional restrictions of the Google Play Store we were not able to download six of them. In the end, we performed our analysis with 124 companion apps. The Table 5.2 contains further details.

In addition, we selected apps based on the *top selling free* category from the Google Play Store for the comparison with general applications. In the end, we used the following 38 apps listed in Table 5.3 for our analysis.

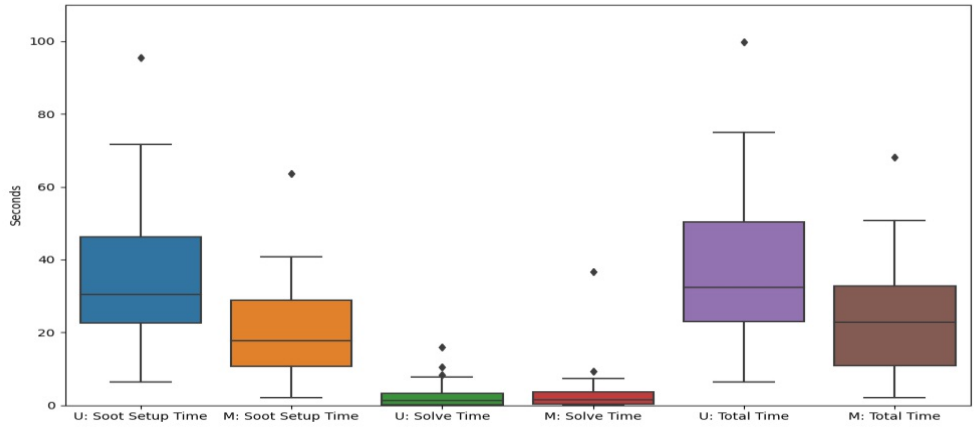


Figure 5.1: Execution times for LeakScope and ValueScope from the same apps of our companion app dataset

The country classification shown in both tables was made based on the developer information from the Google Play Store. If there were location information available, we used those for querying the country from Google Maps. When the location information was not available, we used whois to get the country from the provided website domain. If there was neither a location nor a website provided, we queried the country based on the email address's domain.

## 5.2 Comparison of LeakScope and ValueScope

To compare our extension of LeakScope to the unmodified analysis, we ran LeakScope on the same companion apps with the same configuration as ValueScope. From the 124 companion apps, the original analysis terminated correctly for 43 of them. To make it comparable, we only used the results from apps both implementations terminated without any error. Since our implementation only failed to analyze two apps, it would be otherwise not comparable. From the 43 apps, both analyzes terminated correctly. They reconstructed from 31 apps values. There are different reasons why an analysis is not finding any values. For example, an app might not use any functions we are looking for, or the app is obfuscated, and therefore the analysis cannot reconstruct any values.

In Figure 5.1 we show for both analysis the execution times. The *U:* in the diagram stands for the *"unmodified analysis"* of LeakScope and the *M:* for our *"extended and modified implementation,"* ValueScope. The *setup time* is the time needed for the analysis to initialize Soot. Among other things, Soot needs to load and translated the byte code into the Jimple intermediate representation during this phase. The *solve time* represents the time needed to backtrack and later reconstruct the values. The last columns show the *total execution time*, which are the Soot setup time and the solve time combined.

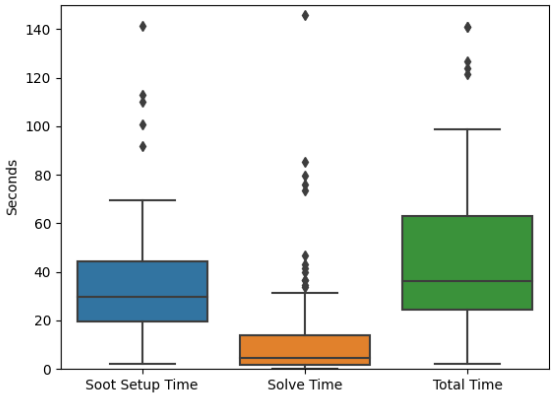


Figure 5.2: Execution times of ValueScope from all 122 companion apps

Besides the execution times, we compared the found domains from both implementations. We therefore first extracted all subdomains from the results with regular expressions. ValueScope was able to extract 93 different subdomains from the 31 apps which contained results. LeakScope found 66 unique subdomains from those apps. We further compared the unique subdomains found in each app. In such a comparison, our extension found in total 33 subdomains more. The following example explains why there is a difference between the per-app and the overall comparison: One implementation might find in the first and second app followings domains *{"google.com", "facebook.com"}*, the other one might find for the first app *{"google.com"}* and for the second *{"facebook.com"}*. In an overall comparison, both implementations have found the same amount of domains, but the first implementation has found two more on a per-app sight.

We later tried to remove invalid findings by resolving the subdomain. Nevertheless, there were also valid subdomains removed during this step, *espressif.cn* is one of them. If we search for the domain in Google, it seems to be the Chinese domain for Expressif devices while the international domain is *espressif.com*. Comparing the filtered results, ValueScope found 79 valid subdomains and LeakScope 54. The total per-app difference is 26.

## 5.3 ValueScope Results

From the 124 companion applications we analyzed with ValueScope, 122 successfully terminated. From the 122 apps successfully executed, we found in 110 of them reconstructed values. The results contained 948 unique subdomains. From those, we were able to look up the IP addresses for 646 of them. In total, they belong to 252 different domains. On a per-app basis, we found on average 12 different subdomains which belong to 4 domains in average. For 36 apps, our analysis detected some form of local communication. We


Figure 5.3: Word cloud of all local IP addresses found

visualized the local IP addresses found in Figure 5.3. The word cloud also contains the string *fromUI.local* which stands for addresses inputted from the user. In total, our analysis found six apps containing addresses from user input. We manually analyzed those apps and found out that, indeed, all of them ask the user to input or select the IP address of the corresponding device.

In Figure 5.2 we provided detailed information about the execution time of ValueScope. This time the diagram shows the execution times of all 122 apps we successfully analyzed. In total, there are even more outlier further outside the represented space. We decided to cut the diagram at 150 seconds of to keep it better readable. From the apps, we analyzed the longest runtimes we encountered was for the setup time 2:46 minutes (166.88 seconds), for the solve time 12:01 minutes (721.95 seconds), and for the total computation time 12:52 minutes (772.68 seconds).

### 5.3.1 Companion Apps and General Popular Apps

In Figure 5.4 and Figure 5.5 we provide two maps which show the domain locations. The first map contains the domain locations we found while analyzing the general apps from Table 5.3. The second map contains the domains found for the companion apps. To create the maps, we first extracted the unique subdomains per app. Later we resolved the subdomains and queried the country of the IP address location with whois queries. We decided to perform the evaluation based on the subdomains since there are sometimes subdomains of the same domain in different countries. For example, within a Xiaomi app, we found the following domains *cn.register.xmpush.xiaomi.com* and *ru.register.xmpush.global.xiaomi.com*. If we only look up *xiaomi.com* we would think that both are located in China. By querying the subdomains, we find out that one is located in Russia and the other one in China. Other examples are *amazonaws* subdomains. In addition, we provided two pie charts in Figure 5.6, which show the


Figure 5.4: Subdomain locations from all general popular apps we analyzed

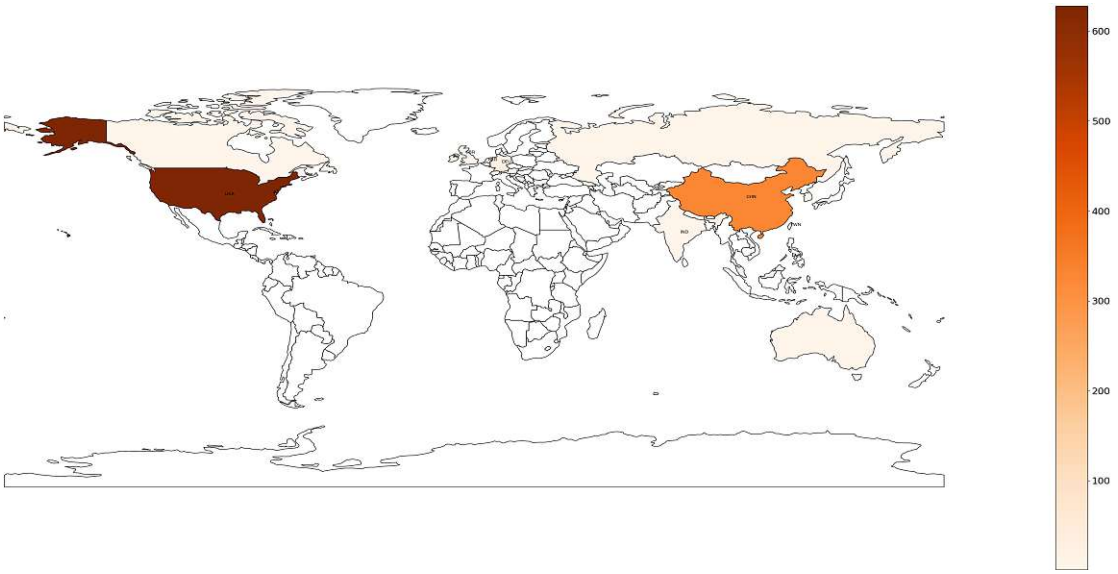


Figure 5.5: Subdomain locations from all companion apps we analyzed

distribution of subdomain locations. For the pie charts, we summarized all countries from the European Union as EU. Since the United Kingdom left the EU on 31.01.2020, we added those among others to the *"Other"* category.

In addition, we visualized the domains from both datasets in Figure 5.7. In that Figure, a connected graph is displayed, making the overlapping domains from both datasets visible. In order to keep the graph clear, we decided not to distinguish different subdomains and

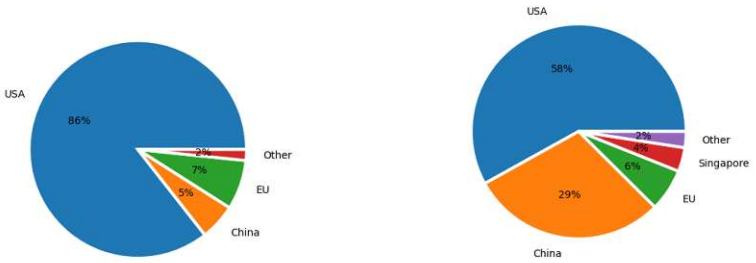


Figure 5.6: The chart on the left-hand shows the distribution of subdomain location from all general popular apps. The chart on the right-hand side shows the distribution of subdomain location from all companion apps we analyzed

handle them as one domain. We increased the size of the domain node by one for each unique subdomain occurring in an app. That means if the subdomains from our previous example (*cn.register.xmpush.xiaomi.com* and *ru.register.xmpush.global.xiaomi.com*) occur in one app, we set the size of the *xiaomi.com* node to two. If those two subdomains occur in another app too, the node's size is set to four. However, if the same subdomain occurs multiple times within an app, it is only counted once. For the size of the *companion app* and *general app* node, it is the same. Both represent the number of analyzed apps from those categories. Also, we categorized the domains into three categories for which we are not showing the single domains. The categories are as follows: *Advertisement and Tracking*, *Content Distribution Networks* and *Social Networks*. In the end, we added a fourth category, *other domains*, that contains all domains occurring less than three times.

### 5.3.2 Companion Apps

For a better understanding of the companion apps we analyzed, we decided to show another connected graph. The graph is provided in Figure 5.9. It shows which domains are called by an app and which domains share some similarities by connecting to the same domains. To better see the connections of the non-standard domains, we are not showing the domains categorized in one of the three categories *Advertisement and Tracking*, *Content Distribution Networks* and *Social Networks*. In addition, we are not showing domains occurring less than three times. The size of the domain nodes again depends on the occurrences of the subdomains. The graph in Figure 5.8 shows the Xiaomi cluster in more detail. In comparison to the overall Figure 5.9, the domains occurring twice are also shown here.

We also evaluated which schemes were used within the different URLs found by ValueScope. We provided the schemes, amounts of value points, the amounts of apps, and the percentage of apps where we found the scheme in Table 5.1. The amount column represents the number of value points where the scheme was found. Value points are statements which are matching one of the signatures for which we were searching. The amount of the found schemes is increased for each value point where it is found. For

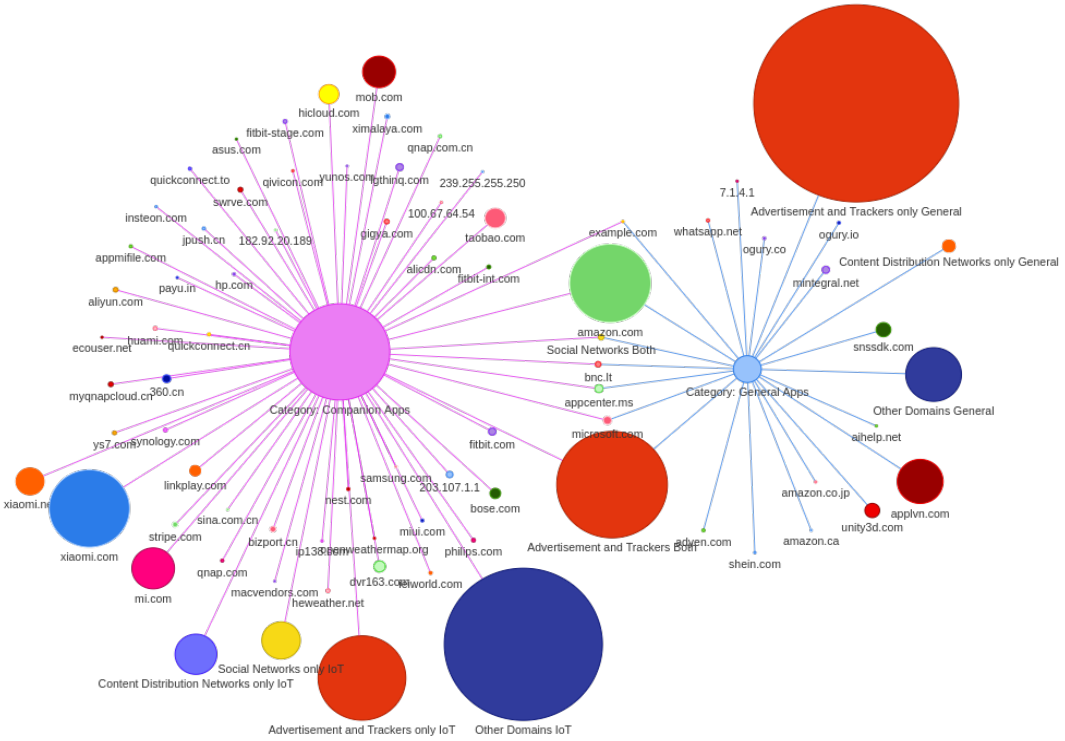


Figure 5.7: Domains from general popular apps and companion apps

example, if we find in a value point twice *http* and once *https*, we increase the amount of both schemes by one. The app row symbolizes the number of apps in which we have found the scheme at least once. The percentage is the relation between the apps in which we have found the scheme and the total number of apps we have found from the respective category. For the *other connection* category, we found 108 apps containing at least one scheme, and for the *local connection* category, we found in 30 apps schemes. We can only say for sure that a scheme is used for local communication if it occurs in an URL with a local IP address or our fall-back value `fromUI.local` for user-inputted domains or IP addresses. Still, there are many cases where we could not reconstruct the host. For example, if it is saved in the shared preferences. Therefore, a scenario is that the hardcoded or user-selected host address is only used for the first connection. Later, the address is saved in the shared preferences. For further local connections, the value from the shared preferences is taken. In such a case, the host is not reconstructed. We cannot say that it is a local communication even if it is used for such in reality. Consequently, we count it to the other values. A value point can also contain multiple schemes. For example, if it is found within a static method. The same comes true for the local and "other" category. If, for one value of a value point, a scheme combined with a local address is found, we count it to the local communications. However, if another

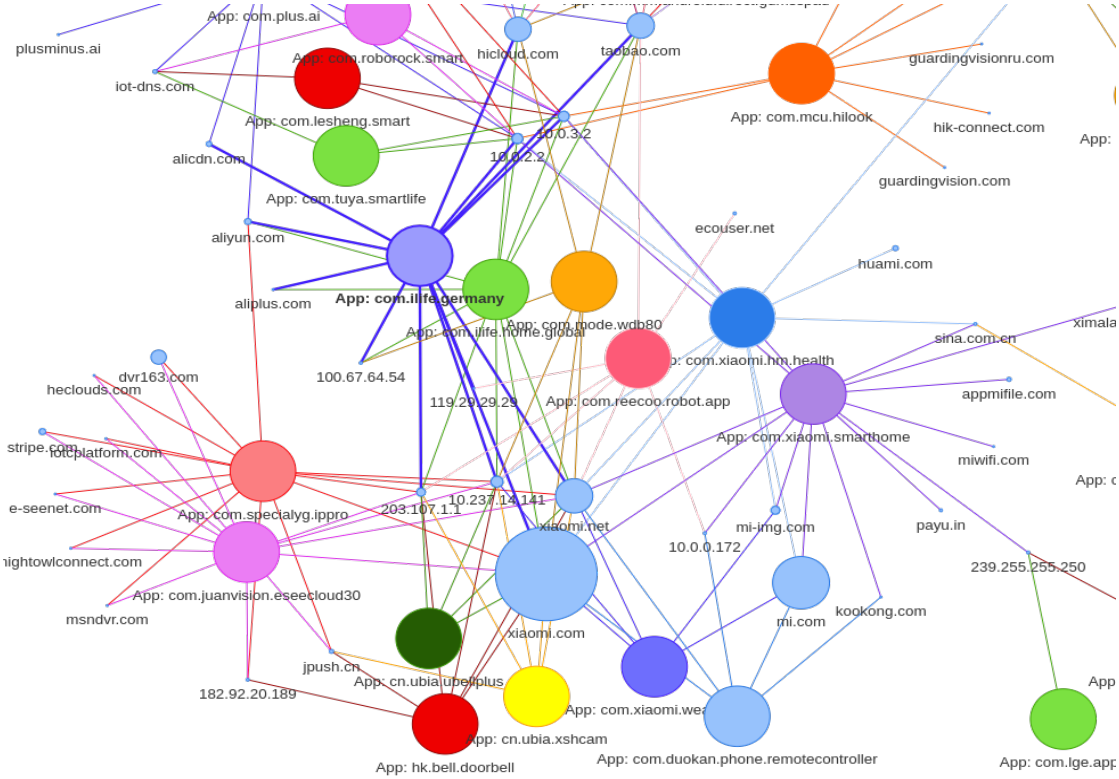


Figure 5.8: A detailed fraction from the Xiaomi cluster

| Other Connection | | | | Local Connection | | | |
|---|---|---|---|---|---|---|---|
| **Scheme** | **Amount** | **Apps** | **Percentage of Apps** | **Scheme** | **Amount** | **Apps** | **Percentage of Apps** |
| https | 3487 | 100 | 92.59 % | http | 420 | 29 | 96.67 % |
| http | 6612 | 87 | 80.56 % | https | 75 | 2 | 6.67 % |
| other | 70 | 17 | 15.74 % | other | 1 | 1 | 3.33 % |
| ws | 54 | 11 | 10.19 % | | | | |
| file | 40 | 7 | 6.48 % | | | | |
| jar | 94 | 5 | 4.63 % | | | | |
| smb | 105 | 3 | 2.78 % | | | | |
| wss | 5 | 3 | 2.78 % | | | | |

Table 5.1: All schemes we found in our results

value is found where we cannot be sure that it is used for local communication, it is counted for the other category. The *other* schemes contain default values we used for reconstructing Java *URL* and *URI* objects. For both, we have problems if the protocol is not matching one of the supported ones. In this case, we need to set default values to get still results for the already reconstructed parts.

Another analysis we performed based on the results reconstructed by ValueScope, was a search for privacy related keywords within the reconstructed query parameters. In

total, we found 27 values belonging to the following keywords: *password, SSID, phone, IMEI, email, latitude, longitude, phonenumber, IP, pass, hostname, mac, secret*. We later discuss two found cases in more detail. Although, most of the found keys belong to intended flows for setting up the devices or authentication.

Besides, we found in 2 apps firmware update URLs from which we were able to download the file. However, for other firmware updated URLs found, we need additional parameters like the current firmware version or the device serial number, which were statically not reconstructed.

## 5.4 IoTFlow

As for ValueScope we executed IoTFlow on the collected companion app dataset. For 113 of them, we were able to collect results. The average runtime per app was 6:30 minutes. In total, there were 255 flows found. Nevertheless, most are less relevant, and some are redundant. Therefore, we provide from two selected apps the flows we found in detail. We selected those apps since they contain the most relevant flows found by our approach.

The first interesting flows we want to present were found in the *com.wifiaudio.Belkin* app. We found there flows from Wi-Fi scanning, from accessing location information, and local communication. In addition, ValueScope was able to find additional information about the endpoints.

In the first flow, we found that the Wi-Fi SSID is read and later sent. Furthermore, ValueScope reconstructed the following URL `https://10.10.10.254/httpapi.asp?command=` `↪ ConnectMasterAp:ssid=NOT_FOUND:ch=NOT_FOUND:auth=NOT_FOUND:encry=NONE:pwd=:chext` `↪ =0`.

In another flow, the longitude and latitude are read and later sent. Again ValueScope provides additional information about the endpoint the data is sent to `https://10.10.10.254/` `↪ httpapi.asp?command=setTuneinLocation:latitude=:longitude=:serial=`.

The last flow to mention here is between two network statements, one flagged as used locally. Since the network functionalities are handled within *util* classes used for local and remote connection, it is not providing additional information. Nevertheless, the reconstructed values from ValueScope have found lots of further commands for the device and cloud endpoints.

In *com.sercomm.gaia.platform.smarthome* the second app, we found a flow where the IMEI (a unique phone identifier) of the phone is read and sent. The reconstructed path, from IoTFlow, gives us the information that the data is sent during the logout process. Since the data flow is found within a non obfuscated method called logout. In this case, ValueScope is not providing further information since the corresponding URL originates from a value saved in the shared preferences. Consequently, we cannot reconstruct it statically.

| APP ID | Country | Version Number | APP ID | Country | Version Number |
|---|---|---|---|---|---|
| com.allegion.leopard | USA | 3.1.0 | com.enphaseenergy.myenlighten | USA | 3.5.9 |
| com.amazon.dee.app | USA | 2.2.398067.0 | de.sma.energy | DEU | 1.03.108.R |
| com.asus.aiextender | TWN | 1.0.0.1.36 | com.eq_3.max_eq3 | DEU | 3.2.1 |
| com.asus.aihome | TWN | 1.0.0.6.21 | de.eq3.pscc.android | DEU | 2.6.14 |
| com.asustek.aicloud | TWN | 2.1.0.0.94 | com.hager.mood | DEU | 1.9.49 |
| com.bitaxon.app.ew001.wizard.freebuds | CHN | 2.6 | com.hager.koala.android | DEU | 3.8.2 |
| com.bose.bosehear | USA | 1.5.4 | com.espressif.rainmaker | CHN | 2.2.4 - 5724bef |
| com.bose.controlspace.remotecustom | USA | 2.9.0.32554 | com.espressif.iot | CHN | v1.2.5 |
| com.bose.corporation.bosesleep | USA | 3.0.7 | h5.espressif.esp32 | CHN | 1.2.2 |
| com.bose.monet | USA | 15.0 | com.espressif.provbleavs | CHN | 2.1.1 |
| com.bose.soundtouch | USA | 26.0.3 | com.xiaomi.smarthome | CHN | 6.4.701 |
| com.cisco.connect.cloud | USA | 2.15.5 | com.xiaomi.wearable | CHN | 2.6.3i |
| com.dlna.asus2 | TWN | 2.0.0.2.81 | com.xiaomi.hm.health | USA | 5.0.1 |
| com.dragonflow | USA | 3.1.78 | com.duokan.phone.remotecontroller | USA | 6.0.4G |
| com.fitbit.FitbitMobile | USA | 3.39.2 | com.sercomm.app.ipcamera | USA | V2.0.6.0 |
| com.fridaylabs.fridaylock | USA | 1.0.14 | com.sercomm.gaia.platform.smarthome | USA | 1.0.1274 |
| com.huawei.bone | CHN | 21.0.1.307 | com.sercomm.gaia.tulip | USA | 0.1.9 |
| com.huawei.ch100 | CHN | V1.1.11.120 | com.zte.linkpro | CHN | V5.2.4.020 |
| com.huawei.colorbands | CHN | 1.3.7.128 | com.hp.android.printservice | USA | 21.3.52 |
| com.huawei.overseas_ah100 | CHN | V1.1.7.120 | com.synology.DSfile | TWN | 4.13.1 |
| com.juanvision.eseecloud30 | USA | 3.3.33 | com.synology.DSfinder | TWN | 2.3.3 |
| com.specialyg.ippro | USA | 3.3.33 | com.synology.dsdrive | TWN | 2.3.0 |
| com.insteon.insteon3 | USA | 1.9.8 | com.synology.dsvideo | TWN | 3.4.3 |
| com.my.leo.switchcontroller | USA | 151204a | com.mcu.iVMS | CHN | 4.7.7 |
| cn.ubia.xshcam | CHN | 1.1.2 | com.mcu.hilook | CHN | 3.10.1.0924 |
| hk.bell.doorbell | HKG | 1.0.182 | com.mcu.iVMSHD | CHN | 4.1.3 |
| cn.ubia.ubellplus | CHN | 1.0.11 | com.hikvision.HikCentralHD | CHN | 1.5.0 |
| com.hogarcontrols.hogarcamhd3.gcm | USA | 2.2.2.70 | com.mm.android.direct.gdmssphone | CHN | 4.90.000 |
| com.pg.oralb.oralbapp | USA | 8.3.1 | com.mm.android.direct.gdmssphoneLite | CHN | 3.53.001 |
| com.philips.ka.oneka.app | CAN | 7.4.0 | com.mm.android.direct.gdmsspad | CHN | 4.00.000 |
| com.philips.lighting.hue2 | USA | 3.48.2 | com.amazon.storm.lightning.client.aosp | USA | 2.1.2172.0-aosp |
| com.signify.hue.blue | USA | 1.31.0 | com.oplayer.silvercrest | USA | V1.0.33 |
| com.tpvision.philipstvapp2 | NLD | 2.1.38 | com.veepoo.hband | CHN | 6.2.5 |
| com.philips.cdp.ohc.tuscany | CAN | 10.0.0 | com.jaga.ibraceletplus.smartwristband | USA | 3.6.5 |
| com.rcreations.ipcamviewerBasic | USA | 7.3.0 | com.xman.tecno.watch | CHN | 1.9.2.27 |
| com.samsung.android.oneconnect | KOR | 1.7.64.21 | com.qihoo.smarthome | USA | 7.3.5.0 |
| com.samsung.dockingaudio2.phone | KOR | 5.0.3 | com.lesheng.smart | CHN | 1.1.6 |
| com.samsung.washer | KOR | 2.1.40 | com.yugong.Backome | CHN | 5.1.2 |
| com.sierrawireless.mhswatcher | USA | 7.16.2004.195 | com.irobot.home | USA | 5.4.0-release |
| com.mode.wdb80 | USA | 3.0.2 | com.roborock.smart | CHN | 2.3.46 |
| com.sonos.acr | USA | 11.2.6 | com.reecoo.robot.app | SGP | 1.3.2 |
| com.thetileapp.tile | USA | 2.75.0 | de.flole.xiaomi | USA | 1.2475-gpe |
| com.tplink.omada | HKG | 3.2.8 | com.eufylife.smarthome | USA | 2.5.70 |
| com.tplink.skylight | HKG | 3.1.18 | com.ilife.germany | HKG | 4.0.17 |
| com.tplink.tpplc | HKG | 1.3.5 | com.ilife.home.global | HKG | 1.2.3 |
| com.tplink.tpmifi | HKG | 2.1.1 | it.positec.landroid | USA | 2.0.3 |
| huiyan.p2pwificam.client | USA | 1.3.25 | com.qnap.qfile | USA | 2.10.8.0218 |
| wansview.p2pwificam.client | USA | 1.0.18 | com.qnap.qvideo | USA | 3.10.13.0111 |
| com.microsoft.xcloud | USA | 1.12.2102.0401 | com.mm.android.direct.gdmsspadLite | CHN | 3.60.001 |
| com.osstream.xboxOneController | ISR | 1.56 | br.com.amt.v2 | BRA | 3.32 |
| com.playstation.remoteplay | USA | 4.1.0 | br.com.intelbras.mibocam | BRA | 1.2.1 |
| com.playstation.mobile2ndscreen | USA | 21.3.1 | epson.print | JPN | 7.6.4 |
| com.nest.android | USA | 5.61.0.2 | com.brother.mfc.brprint | USA | 6.5.0 |
| com.belkin.android.androidbelkinnetcam | USA | 2.0.5 | com.ricoh.smartdeviceconnector | JPN | 3.14.2 |
| com.belkin.wemoandroid | USA | 1.29.1 | com.primax.MobileSDC220 | JPN | 1.06 |
| com.wifiaudio.Belkin | USA | 1.0.1.200925 | com.lge.app1 | USA | 5.0.5 |
| com.belkin.btapp | USA | 1.0.2 | com.lgeha.nuts | USA | 3.5.1722 |
| com.philips.src.hss | CAN | 3.10.1 | com.plus.ai | USA | 3.3.2 |
| com.superthomaslab.hueessentials | NLD | 1.21.0 | de.telekom.smarthomeb2c | DEU | 6.3.0_8d03a177 |
| com.ikea.tradfri.lighting | USA | 1.14.2 | com.dlink.mydlinkunified | TWN | 2.3.1 |
| com.philips.vitaskin.male | CAN | 7.1.0 | com.dlink.mydlinkmyhome | TWN | 3.0.11 |
| com.ecobee.athenamobile | CAN | 8.7+134947 | com.tuya.smartlife | HKG | 3.26.5 |

Table 5.2: The list of companion apps we used for our analysis

| APP ID | Country | Version Number | APP ID | Country | Version Number |
|---|---|---|---|---|---|
| com.instagram.android | USA | 54.0.0.14.82 | com.contextlogic.wish | USA | 4.47.5 |
| com.uncosoft.highheels | USA | 1.5.1 | com.teacapps.barcodescanner | DEU | 2.6.9-L |
| cn.danatech.xingseus | CHN | 2.12 | com.roblox.client | USA | 2.472.420209 |
| com.whitesquare.animaltransform | USA | 0.6.1 | com.zhiliaoapp.musically | SGP | 18.9.5 |
| com.innersloth.spacemafia | USA | 2021.4.2 | com.fivebits.emeregencydispatch | ISR | 1.065 |
| com.disney.disneyplus | USA | 1.14.1 | com.smo.deepcleaninc3d | TUR | 1.0.34_a |
| com.smallgiantgames.combat | FIN | 31.1.0 | com.barsstudios.swordplay | ARE | 2.5 |
| com.playstrom.bob | BLR | 1.1.1 | com.DefaultCompany.CatNDog | USA | 1.2.5 |
| com.shopify.arrive | CAN | 2.21.0 | com.amazon.avod.thirdpartyclient | USA | 3.0.293.4645 |
| com.Garawell.BridgeRace | ISR | 2.026 | com.king.crash | MLT | 1.0.81 |
| com.Pizia.VoodooDoll | USA | 0.45 | com.paradyme.solarsmash | GBR | 1.4.1 |
| com.gma.water.sort.puzzle | AUS | 3.0.7 | com.game.colorslime | TUR | 1.49 |
| com.zzkko | USA | 7.5.2 | com.snapchat.android | USA | 11.23.2.36 |
| com.boltrend.disgaea.en | HKG | 1.0.254 | com.casual.impostor.smasher | SGP | 1.0.8 |
| us.zoom.videomeetings | USA | 5.6.0.1592 | com.google.android.play.games | USA | 2021.02.24918 |
| com.paypal.android.p2pmobile | USA | 7.39.2 | com.elex.twdsaw.gp | HKG | 1.2.3 |
| com.crazylabs.shoal.of.fish | ISR | 0.0.3 | com.discord | USA | 69.0 |
| org.telegram.messenger | ARE | 7.6.0 | com.crazylabs.diy.make.up | ISR | 1.1.1 |
| com.amazon.mShop.android.shopping | USA | 22.7.0.100 | com.whatsapp | USA | 2.21.7.14 |

Table 5.3: List of generally widely used applications we used for our analysis

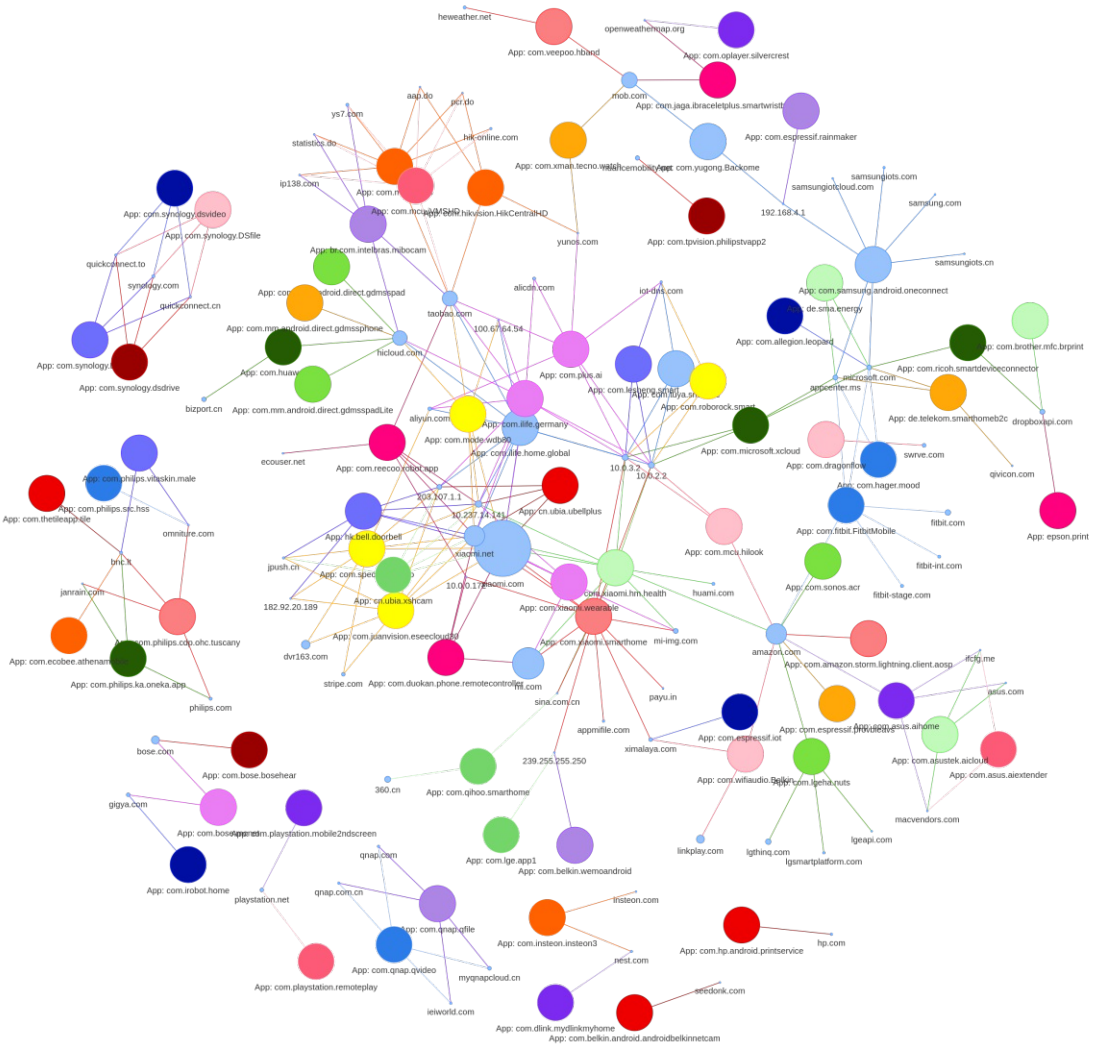


Figure 5.9: Connected graph from the companion app datasets

# CHAPTER 6

# Discussion

In the following sections, we discuss the results presented in the previous chapter. We also show the limitations of our approaches and give a prospect of our future work.

## 6.1 Observations

### 6.1.1 Comparison of LeakScope and ValueScope

While comparing ValueScope and LeakScope, the question arises why LeakScope and ValueScope do not report any results for 81 and two apps, respectively. Our extended analysis runs out of memory for two apps, and therefore the analysis is aborted. These apps are *com.amazon.dee.app* and *com.ikea.tradfri.lighting*. For the original LeakScope implementation, we identified three reasons why the analysis is not terminating correctly. First, it also runs out of the heap space. In addition, the analysis is often aborted because runtime exceptions are thrown, which are never caught. The third reason is that a timeout gets triggered for some apps, and the analysis is aborted without gathering any intermediate results.

Comparing the execution times in Figure 5.1, we can see that ValueScope is faster on average during the setup. That leads to a faster overall analysis. The main reason for the faster setup time is that we excluded more libraries that are not relevant for the analysis. The exclusion of libraries allows us to load less code and therefore speed up the complete analysis. Taking a closer look at the solve time, we can see that ValueScope takes slightly longer on average. In addition, the outliers of ValueScope are far higher than the ones from LeakScope. The reason for this is that we are modeling more network-related methods. For example, if the following statement `<okhttp3.Request$Builder: okhttp3.Request$Builder url(okhttp3.HttpUrl)>` is found by LeakScope, the analysis cannot backtrack it and therefore cannot compute any values. LeakScope cannot analyze the statement since it is not aware of how to handle `okhttp3.HttpUrl`. In comparison, our

extended version does handle and analyze it. Still, as we can see in Figure 5.1, the speed up during the setup time outweighs this and results in an overall speedup.

Both implementations did not find any results for the same 12 apps. They do not find results because apps are obfuscated, using native code for network functionalities, and one app was not using any network functionalities at all. The differences of the found domains and subdomains result from the additionally handled objects like the `okhttp3.HttpUrl`. Comparing the domains before and after the validation step, around 85 % of the subdomains found by ValueScope were valid. For LeakScope, around 82 % were valid. The reason for the higher ratio can again be attributed to the additional handled network objects. Compared to `java.net.URL` which can also be used for opening files, the one from okhttp is only used for actual requests. Therefore, it is more likely that a reconstructed URL contains a valid domain. For example, in the *com.primax.MobileSDC220* app we found `file:/ByteArrayClassPath/float.class` and similar URLs. These are used to load code dynamically. Both analyzes reported these URLs, but since they do not contain any valid domain, they are discarded during the validation step at the latest.

### 6.1.2 ValueScope

Looking at the Figure 5.2, it is similar to the one previously shown for the subset of companion apps. As for the subset, the analysis spends more time setting up Soot than actually analyzing the code. Comparing the times spent during the different analysis phases, they are now higher. We assume that the reason is that previously only apps were selected where the original implementation was also terminating correctly. As explained above, LeakScope terminates the analysis if it is taking too long without collecting any results. In addition, it is more likely that a large app contains statements triggering a runtime exception that LeakScope cannot handle. For the same reason, the 12 apps where the analysis did not find any results are included in the smaller subset. These are much simpler for the analysis to handle. If there are no methods found to trace, it is unlikely that the analysis runs into time issues or that exceptions are thrown.

Next, we take a look at the local IP addresses. The three local addresses we most often found are *10.0.2.2*, *10.0.3.2* and *10.237.14.141*. The first two addresses are false positives. Such values can be used as aliases to access resources on the host while working on an emulator. Therefore, they can also be used for emulator detection. Some apps try to detect if they are running on an emulator to make dynamic reverse engineering harder. We manually investigated the apps where we found *10.0.2.2*, and *10.0.3.2*. All occurrences of these addresses were within the "React Native" library by Facebook.[1] A library method will return these IP addresses if it detects that the app is running on an emulator to avoid errors during the development process.

To answer our questions from Chapter 3, we examine the domain locations visualized in the Figures 5.4, 5.5, and 5.6. Most apps connect to servers in the US from both datasets. The second most connected to country is China. Setting the shares in relation to where

[1] React Native library, `https://github.com/facebook/react-native`, accessed: 07.05.2021

the apps were developed, we see that the apps developed in the US slightly drop from 47 % in the general data set to 45 % in the IoT specific one. However, the domains we found from the US drop significantly from 86 % to 58 %. On the other hand, the share of apps developed by Chinese developers (7.9 % in the general apps to 28.2 %) behaves similarly to the share found for the domains belonging to China. Moreover, the increase of domains from Singapore is consistent with the share of apps developed there. Around 8 % of the companion apps we analyzed were developed in Taiwan. Nevertheless, we could not find such growth in the contacted domains. For domains found belonging to countries from the European Union, the fractions stayed nearly the same in both datasets. The same applies to the shares of applications developed in the EU. In the general one, around 7.9 % were developed in the EU, compared to around 6.5 % in the companion app specific dataset.

The differences between the domains connected by general popular apps and companion apps are visible in the connected graph in Figure 5.7. There are overlaps mainly on commonly used services like advertisement and trackers, social networks, or services from big companies like Amazon and Microsoft. However, they differ in what additional domains they call. Advertisement and trackers play a more important role in general apps. The general apps we analyzed also contained fewer independent domains compared to the IoT apps. Looking at the domains from companion apps, we can see that the different manufacturers often use their own domains. This behavior is usually not the case for general apps on such a scale. For companion apps, three of the most occurring domains belong to Xiaomi, namely *xiaomi.com*, *xiaomi.net* and *mi.com*. We identified three reasons why this is the case. First, we have analyzed multiple apps from Xiaomi. Second, Xiaomi uses many different subdomains. We found 32 different subdomains for *xiaomi.com*. Third, some devices from other manufacturers also call those domains.

Many of the advertisement and tracking related services we found were located in the US. In contrast, the manufacturer's domains are generally hosted in their own countries. These are the main reasons why the share of domains from the US dropped in the companion app dataset significantly.

In the connected graph in Figure 5.9, it is visible that the applications from the same manufacturer often use the same set of domains. In Figure 5.8, we show apps connecting to Xiaomi domains. A pink node is to the right of the marked node in the graph. Both of them belong to vacuum cleaner robots from two different brands *com.reecoo.robot.app* and *com.ilife.germany*. Interestingly, both connect to similar domains, which indicates a rebranded device or the use of similar components.

From the Table 5.1 we can infer that *http* without TLS is still widely used in the latest companion apps. However, we cannot say for how many connections and data transmission it is really used. Still, it is pretty shocking that most apps at least have a fall-back mechanism for its use. In several apps, we found conditions on whether or not an app will use *https* based on settings or other criteria.

We found out that the proportion of *http* to *https* is even worse for local communication.

Keeping the private key private while the device and firmware might get reversed is difficult. Such enhancements represent additional work for the manufacturers and add more complexity to the development process. Therefore, it is not surprising that the relation of *https* is even worse than for other connections. For web sockets, the ones enhanced with TLS were the minority found in the dataset.

Another interesting point is that we have found 94 value points belonging to five apps where the *jar* scheme occurred. JAR files can be used to load code dynamically and therefore obfuscate the app. We found that all reconstructed *samba* schemes belong to network or storage related companion apps. Considering this, the found cases are not surprising.

As we have mentioned, we found a local connection in 36 apps. However, only 30 apps contain local communication with schemes. The reason, therefore, is that not for all found values, we have reconstructed corresponding schemes. Consequently, there are addresses for which we have not reconstructed any scheme.

Next, we continue with the sensitive keywords found in our search. The first URL we want to discuss is the following `http://fromui.local:not_found/get_camera_params` ↪ `.cgi?user=not_found&pwd=not_found`, found in the *com.insteon.insteon3* app. We assume from the request that it is sent to an IoT device, and some authentication is performed. It is not recommended to send the password in plain text, even if the request is sent over the local network.

The other remarkable finding is that we have found OAuth requests where the so called *client_secret* is hardcoded. We found such hardcoded secrets in three apps from two different manufacturers. OAuth is used for getting an authorization token. First, a code is obtained through GET requests and redirects. Afterward, the code is exchanged for a token. During this step the *client_secret* is needed. If the secret is hardcoded, a malicious app could listen for the responses containing the OAuth code and exchange it before the actual app does it. To avoid having a hardcoded secret, OAuth recommends using Proof Key for Code Exchange (PKCE), where a new value is generated for the secret each time.[2] One of the OAuth requests also contains the IMEI as a request parameter. We discuss a similar case in the next section in detail. Sending the IMEI is a threat to the privacy of users since there is no necessary scenario for sending it. Also, Google's best practices discourage developers from sending hardware identifiers, like the IMEI number.[3]

[2] OAuth2, `https://developer.okta.com/blog/2018/12/13/oauth-2-for-native-and-mobile-apps`, accessed: 16.05.2021

[3] "Best practices for unique identifiers," `https://developer.android.com/training/articles/user-data-ids`, accessed: 10.05.2021

```
1 public interface SensorDataService {
2     @GET("temperature")
3     Call<SensorData> getTemperature();
4 }
```

Listing 6.1: Sample Retrofit annotation usage

### 6.1.3 IoTFlow

We start with the results obtained from the Belkin app. Even if both flows that send data from the phone to the local device represent intended functionality, they help us to get a further understanding of the IoT device. Furthermore, the reconstructed URLs with the device's commands surely help test the device. For example, they could be used further to fuzz the smart device and therefore help to find vulnerabilities. We did not gather much additional information from the local to remote network flow as the cloud and device endpoints were already provided by ValueScope. Still, we believe that such flows hold great opportunities, and we have planned to improve them further in the future, as we explain in Section 6.3.

The flow where the IMEI is sent out from the smarthome app can be seen as PII leakage. It is bad practice since private phone data is transferred, which is not necessary. For the above functionality, a UUID could be generated and used as well. The IMEI number combined with the authentication can also indicate a backend vulnerability. For example, if the developers assume that the IMEI is always unique, which can be violated by spoofing the IMEI.

## 6.2 Limitation

Each component of the thesis faces its own limitations. For the overall analysis, one must be aware of these to avoid wrong result interpretations.

### 6.2.1 ValueScope

One limitation of ValueScope is that it only limitedly supports loops. During backtracking, the program is not aware of the upper bound of a loop. Therefore, it cannot be analyzed precisely. The block is only visited once to avoid missing loop blocks entirely.

Some Java and Android libraries nowadays use annotations a lot to provide additional features. Listing 6.1 shows an example of how a GET request is performed with the retrofit library.[4] The path for the request is added within the annotation. Later, the library generates the actual class implementation from the interface. With ValueScope, it

[4] "A type-safe HTTP client for Android and Java," https://square.github.io/retrofit/, accessed: 11.04.2021

is not possible to find such annotations. Consequently, the corresponding values cannot be reconstructed at the moment. To handle this issue, the interesting method search of ValueScope needs further changes.

By design, ValueScope can only compute values for functions and objects it is modeling. Consequently, our analysis cannot handle many objects and methods, and therefore no results are found for them.

Another limitation is that the analysis runs out of heap space while analyzing some vast apps. Part of our future work is to implement a memory watcher like FlowDroids to avoid that problem. The memory watcher can help to collect the results and end the analysis before it runs out of memory. For some apps, a trivial solution would also be increasing the heap space for the analysis.

The simple privacy keyword search is only looking for a limited set of words at the moment. It is also only searching for those values in the request keys. That leads to missed values if the keyword is encoded in the request path, for example.

### 6.2.2 IoTFlow

Since IoTFlow is based on FlowDroid and ValueScope, the approach inherent all limitations from those two approaches. Similar to the limitation concerning annotations in ValueScope, IoTFlow can also not handle them. That limits the source finding task for local communications.

Besides this, apps often use static methods to perform web requests. Such static methods can be used to communicate with the IoT device and send data to remote servers. Therefore, IoTFlow might report data flows between remote servers in those cases.

In many cases, we have observed that the flows detected do not help because they are between the same endpoints. For example, data flows between the same Bluetooth connection are reported.

### 6.2.3 Endpoint Evaluation

As described in Chapter 3 we first looked up the IP address of the domains. A real user located in China might query another DNS server and get different results in the end than we do. Additionally, not all countries are represented in the GeoPandas[5] dataset. Consequently, our map does not show them.

A limitation of our comparison between general apps and companion apps is that we selected fewer general popular apps. As a result, the bias of the selected apps is higher, and the overall picture could differ.

To provide the country locations of the app developers, we used different strategies, depending on the information provided in the Google Play Store. We either used

[5]Python GeoPandas library, `https://geopandas.org/`, accessed: 12.04.2021

the location, website, or email address provided by the developers. Nevertheless, this information could point us to other countries than the actual country of the manufacturer. For example, app developers might have multiple versions of the app for different regions. Another case where our country association could be wrong is if only the mail address is provided. For example, suppose that the domain from the email address is from a commonly used mail domain like *gmail.com*. We found such a case once by an unofficial companion app.

## 6.3 Future Work

We already have several plans for future work. First, we want to perform our analysis on a larger scale. To speed up the analysis for single apps, we plan to integrate IoTFlow's local source finding task better into the single tasks performed by FlowDroid. To remove the limitation where flows between the same endpoint are found, we plan to categorize the sources and sinks and only report a flow if they correspond to matching categories. We can then focus on the flows between the smart devices and the cloud with this extension. Another future work is to extend the local source finding to a more general source finding for all found value points. With this extension, we could more reliably report the found values since they are sometimes not reported if the flow left out the statement where the URL is added to the request.

We plan to merge domains from the same company, like the ones found from *xiaomi.com*, *xiaomi.net* and *mi.com*. This improvement can help keep the further evaluation of the called domains clear, even if performed on a larger scale.

From the found *OAuth client_secret*, we got the idea to do further testing regarding the found *OAuth* requests.

The domains we have treated as false positives, which we could not resolve, also raise an opportunity for future work. Pariwono et al. [35] showed the threats from abandoned domains to their user's privacy and their potential to misuse them for scamming or phishing. In the future, it would be interesting to investigate further why the domains are not resolving. Because our static analysis did not reconstruct them correctly or the domains are abandoned and therefore present further threats to the users.

Additionally, we plan to perform our static analysis on different apps and different versions of the same app to gather knowledge about how sending data evolved over time. Such an analysis could give an overview of the influence of the general data protection regulation (GDPR) [16] on IoT privacy. Similarly, it would be interesting to see how the GDPR influences the domain locations called by the companion apps.

Our work shows that many data flows found are not data leaks but belong to intended app traffic. Previous work already faced the problem that not all flows detected by automated analysis techniques are indeed data leaks. Pan et al. [34] tried to identify legitimate data flows by natural language processing to avoid false positives. Such an extension would also be interesting for IoTFlow.

Furthermore, we plan to automatically classify the domains we obtain through ValueScope into first- or third-party. That can help us to decide if a data flow represents a privacy leak or is intended. In addition, we gain knowledge about how serious a leak is since it makes a difference who has access to the personal data.

ValueScope gives us the ability to reconstruct more values and not only URLs. In the future, we want to gather additional information about other sources and sinks.

Another exciting opportunity for the future are further evaluations about how companion apps handle the data concerning obfuscation, encryption, and encoding.

# CHAPTER 7

# Conclusion

Large-scale analysis in the IoT area can be seen as the missing link to improve the privacy and security of smart devices. The importance of a large-scale analysis results from the diversity of the smart devices. Dynamic analysis hardly scales since the actual devices are needed, which results in unaffordable analysis. In addition, a dynamic analysis is time-consuming since it requires extensive reverse engineering for each device. Since the software and hardware of smart devices are diverse, it is hard to develop a general strategy for analysis.

With our work, we are making a step towards large-scale analysis of IoT devices. Our analysis is done statically on companion apps to circumvent the barriers. For the analysis, we identified two main aspects of companion apps that distinguish them from other apps: they connect over the local network to their corresponding devices, and there are other protocols in use.

To identify the communication over the local network, we extended an existing value set analysis to fit the needs for reconstructing URLs of Android apps. We showed that our extension is an improvement regarding the obtained results and the analysis speed. Additionally, the reconstructed endpoints contain valuable information about the IoT devices themselves and the cloud endpoints, as we showed in our results.

We integrated the local network finding analysis in an existing general-purpose flow analysis approach. We also adapted further the sources and sinks to cover different protocols used by IoT devices.

In total, we analyzed 124 companion apps. In addition to the evaluations based on the reconstructed URLs, we presented the results found from our flow analysis for two apps in detail. Our results show the possibilities of finding security and privacy problems with our approach. Furthermore, we highlighted some information obtained by reconstructing URLs of companion apps.

There are still further steps to improve our analysis in order to reduce the manual work, enhance the URL reconstruction, and make further use of the gathered information.

Overall, we are taking a step towards large-scale analysis of PII leakage in IoT companion apps.

APPENDIX A

# Appendix

## A.1 Sources and Sinks Added

```
%------------Source----------%
%-------------BL------------%
<android.bluetooth.BluetoothAdapter: android.bluetooth.BluetoothDevice getRemoteDevice(byte[])
    ↪ > -> _SOURCE_
<android.bluetooth.BluetoothGattService: int getInstanceId()> -> _SOURCE_
<android.bluetooth.OobData: byte[] getLeBluetoothDeviceAddress()> -> _SOURCE_
<android.bluetooth.BluetoothDevice: android.os.ParcelUuid[] getUuids()> -> _SOURCE_
<android.bluetooth.BluetoothAdapter: java.lang.String getAddress()> -> _SOURCE_
<android.bluetooth.BluetoothAdapter: java.util.Set getBondedDevices()> -> _SOURCE_
<android.bluetooth.BluetoothAdapter: java.lang.String getName()> -> _SOURCE_
<android.bluetooth.BluetoothDevice: java.lang.String getAddress()> -> _SOURCE_
<android.bluetooth.BluetoothDevice: java.lang.String getAlias()> -> _SOURCE_
<android.bluetooth.BluetoothDevice: java.lang.String getName()> -> _SOURCE_
<android.bluetooth.BluetoothSocket: java.io.InputStream getInputStream()> -> _SOURCE_
%------------BLE-------------%
<android.bluetooth.le.ScanResult: android.bluetooth.BluetoothDevice getDevice()> -> _SOURCE_
<android.bluetooth.le.ScanRecord: java.lang.String getDeviceName()> -> _SOURCE_
<android.bluetooth.le.ScanResult: android.bluetooth.le.ScanRecord getScanRecord()> -> _SOURCE_
<android.bluetooth.le.ScanResult: int getRssi()> -> _SOURCE_
<android.bluetooth.BluetoothGattService: java.util.UUID getUuid()> -> _SOURCE_
<android.bluetooth.BluetoothGattService: int getInstanceId()> -> _SOURCE_
<android.bluetooth.BluetoothGattService: java.util.List getCharacteristics()> -> _SOURCE_
<android.bluetooth.BluetoothGattService: android.bluetooth.BluetoothGattCharacteristic
    ↪ getCharacteristic(java.util.UUID)> -> _SOURCE_
%----------paho.mqtt---------%
<org.eclipse.paho.client.mqttv3.MqttCallback: void messageArrived(java.lang.String)> ->
    ↪ _SOURCE_
<org.eclipse.paho.mqttv5.client.MqttCallback: void messageArrived(java.lang.String,org.eclipse
    ↪ .paho.mqttv5.common.MqttMessage)> -> _SOURCE_
<org.eclipse.paho.client.mqttv3.IMqttMessageListener: void messageArrived(java.lang.String,org
    ↪ .eclipse.paho.client.mqttv3.MqttMessage)> -> _SOURCE_
<org.eclipse.paho.mqttv5.client.IMqttMessageListener: void messageArrived(java.lang.String,org
    ↪ .eclipse.paho.mqttv5.common.MqttMessage)> -> _SOURCE_
<org.eclipse.paho.client.mqttv3.IMqttToken: org.eclipse.paho.client.mqttv3.internal.wire.
    ↪ MqttWireMessage getResponse()> -> _SOURCE_
<org.eclipse.paho.mqttv5.client.IMqttToken: org.eclipse.paho.mqttv5.common.packet.
    ↪ MqttWireMessage getResponse()> -> _SOURCE_
%-------------NFC------------%
<android.nfc.tech.IsoDep: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.IsoDep: byte[] getHiLayerResponse()> -> _SOURCE_
<android.nfc.tech.IsoDep: byte[] getHistoricalBytes()> -> _SOURCE_
<android.nfc.tech.MifareClassic: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.MifareClassic: byte[] readBlock(int)> -> _SOURCE_
```

```
<android.nfc.tech.MifareUltralight: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.MifareUltralight: byte[] readBlock(int)> -> _SOURCE_
<android.nfc.tech.Ndef: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.Ndef: android.nfc.NdefMessage getNdefMessage()> -> _SOURCE_
<android.nfc.tech.Ndef: android.nfc.NdefMessage getCachedNdefMessage()> -> _SOURCE_
<android.nfc.tech.NdefFormatable: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcA: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcA: byte[] getAtqa()> -> _SOURCE_
<android.nfc.tech.NfcA: short getSak()> -> _SOURCE_
<android.nfc.tech.NfcB: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcB: byte[] getApplicationData()> -> _SOURCE_
<android.nfc.tech.NfcBarcode: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcBarcode: byte[] getBarcode()> -> _SOURCE_
<android.nfc.tech.NfcF: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcF: byte[] getManufacturer()> -> _SOURCE_
<android.nfc.tech.NfcF: byte[] getSystemCode()> -> _SOURCE_
<android.nfc.tech.NfcV: android.nfc.Tag getTag()> -> _SOURCE_
<android.nfc.tech.NfcV: byte getDsfId()> -> _SOURCE_
%-----NFC-Card-Emulation-----%
<android.nfc.cardemulation.HostApduService: byte[] processCommandApdu(byte[],android.os.Bundle
    ↪ )> -> _SOURCE_
<android.nfc.cardemulation.HostApduService: byte[] processCommandApdu(byte[],android.os.Bundle
    ↪ )> -> _SOURCE_
<android.nfc.cardemulation.HostNfcFService: byte[] processNfcFPacket(byte[],android.os.Bundle)
    ↪ > -> _SOURCE_
%---Local Network Scanning---%
<android.net.wifi.WifiManager: java.util.List getScanResults()> -> _SOURCE_
<android.net.wifi.WifiManager: android.net.wifi.WifiInfo getConfiguredNetworks()> -> _SOURCE_
<android.net.wifi.WifiManager: java.util.List getConnectionInfo()> -> _SOURCE_
<java.net.NetworkInterface: java.util.Enumeration getInetAddresses()> -> _SOURCE_
<java.net.NetworkInterface: java.util.Enumeration getInterfaceAddresses()> -> _SOURCE_
%-------------P2P------------%
<android.net.wifi.p2p.WifiP2pManager.ConnectionInfoListener: void onConnectionInfoAvailable(
    ↪ android.net.wifi.p2p.WifiP2pInfo)> -> _SOURCE_
<android.net.wifi.p2p.WifiP2pManager.DeviceInfoListener: void onDeviceInfoAvailable(android.
    ↪ net.wifi.p2p.WifiP2pDevice)> -> _SOURCE_
<android.net.wifi.p2p.WifiP2pManager.GroupInfoListener: void onGroupInfoAvailable(android.net.
    ↪ wifi.p2p.WifiP2pGroup)> -> _SOURCE_
<android.net.wifi.p2p.WifiP2pManager.NetworkInfoListener: void onNetworkInfoAvailable(android.
    ↪ net.wifi.p2p.NetworkInfo)> -> _SOURCE_
<android.net.wifi.p2p.WifiP2pManager.PeerListListener: void onPeersAvailable(android.net.wifi.
    ↪ p2p.WifiP2pDeviceList)> -> _SOURCE_
<android.net.wifi.p2p.WifiP2pManager.UpnpServiceResponseListener: void onUpnpServiceAvailable(
    ↪ java.util.List,android.net.wifi.p2p.WifiP2pDevice)> -> _SOURCE_

%------------SINK------------%
%-------------BLE------------%
<android.bluetooth.BluetoothGatt: boolean writeCharacteristic(android.bluetooth.
    ↪ BluetoothGattCharacteristic)> -> _SINK_
<android.bluetooth.BluetoothGatt: boolean writeDescriptor(android.bluetooth.
    ↪ BluetoothGattDescriptor)> -> _SINK_
%-----------Apache-----------%
<org.apache.http.impl.client.DefaultHttpClient: org.apache.http.HttpResponse execute(org.
    ↪ apache.http.client.methods.HttpUriRequest)> -> _SINK_
<org.apache.http.client.HttpClient: org.apache.http.HttpResponse execute(org.apache.http.
    ↪ client.methods.HttpUriRequest)> -> _SINK_
%-----------OkHttp-----------%
<okhttp3.Call: okhttp3.Response execute()> -> _SINK_
<okhttp3.Call: void enqueue(okhttp3.Callback)> -> _SINK_
%---------paho.mqtt----------%
<org.eclipse.paho.android.service.MqttAndroidClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,byte[],int,boolean)> -> _SINK_
<org.eclipse.paho.android.service.MqttAndroidClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,byte[],int,boolean,java.lang.Object,org.
    ↪ eclipse.paho.client.mqttv3.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.android.service.MqttAndroidClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,org.eclipse.paho.client.mqttv3.MqttMessage
    ↪ )> -> _SINK_
<org.eclipse.paho.android.service.MqttAndroidClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,org.eclipse.paho.client.mqttv3.MqttMessage
    ↪ ,java.lang.Object,org.eclipse.paho.client.mqttv3.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.client.mqttv3.MqttAsyncClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,byte[],int,boolean)> -> _SINK_
```

```
<org.eclipse.paho.client.mqttv3.MqttAsyncClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,byte[],int,boolean,java.lang.Object,org.
    ↪ eclipse.paho.client.mqttv3.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.client.mqttv3.MqttAsyncClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,org.eclipse.paho.client.mqttv3.MqttMessage
    ↪ )> -> _SINK_
<org.eclipse.paho.client.mqttv3.MqttAsyncClient: org.eclipse.paho.client.mqttv3.
    ↪ IMqttDeliveryToken publish(java.lang.String,org.eclipse.paho.client.mqttv3.MqttMessage
    ↪ ,java.lang.Object,org.eclipse.paho.client.mqttv3.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttAsyncClient: org.eclipse.paho.mqttv5.client.IMqttToken
    ↪ publish(java.lang.String,byte[],int,boolean)> -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttAsyncClient: org.eclipse.paho.mqttv5.client.IMqttToken
    ↪ publish(java.lang.String,byte[],int,boolean,java.lang.Object,org.eclipse.paho.mqttv5.
    ↪ client.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttAsyncClient: org.eclipse.paho.mqttv5.client.IMqttToken
    ↪ publish(java.lang.String,org.eclipse.paho.mqttv5.client.MqttMessage)> -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttAsyncClient: org.eclipse.paho.mqttv5.client.IMqttToken
    ↪ publish(java.lang.String,org.eclipse.paho.mqttv5.client.MqttMessage,java.lang.Object,
    ↪ org.eclipse.paho.mqttv5.client.IMqttActionListener)> -> _SINK_
<org.eclipse.paho.client.mqttv3.MqttClient: void publish(java.lang.String,byte[],int,boolean)>
    ↪  -> _SINK_
<org.eclipse.paho.client.mqttv3.MqttClient: void publish(java.lang.String,org.eclipse.paho.
    ↪ client.mqttv3.MqttMessage)> -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttClient: void publish(java.lang.String,byte[],int,boolean)>
    ↪  -> _SINK_
<org.eclipse.paho.mqttv5.client.MqttClient: void publish(java.lang.String,org.eclipse.paho.
    ↪ mqttv5.common.MqttMessage)> -> _SINK_
%------------NFC----------------%
<android.nfc.tech.MifareClassic: void writeBlock(int,byte[])> -> _SINK_
<android.nfc.tech.MifareUltralight: void writeBlock(int,byte[])> -> _SINK_
<android.nfc.tech.Ndef: void writeNdefMessage(android.nfc.NdefMessage)> -> _SINK_
%--------NFC-Card-Emulation-----%
<android.nfc.cardemulation.HostNfcFService: void sendResponsePacket(byte[])> -> _SINK_
<android.nfc.cardemulation.HostApduService: void sendResponseApdu(byte[])> -> _SINK_
<android.nfc.cardemulation.HostApduService: void sendResponseApdu(byte[])> -> _SINK_

%------------Both------------%
%------------NFC------------%
<android.nfc.tech.NfcV: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.NfcF: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.NfcB: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.NfcA: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.MifareClassic: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.IsoDep: byte[] transceive(byte[])> -> _BOTH_
<android.nfc.tech.MifareUltralight: byte[] transceive(byte[])> -> _BOTH_
```

# List of Figures

# List of Tables

# Bibliography


[1] O. Alrawi et al. “SoK: Security Evaluation of Home-Based IoT Deployments”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1362–1380.

[2] *Application Fundamentals*. Accessed: 2020-02-06. 2021. URL: https://developer.android.com/guide/components/fundamentals.

[3] Steven Arzt. “Static Data Flow Analysis for Android Applications”. PhD thesis. Darmstadt: Technische Universität, 2017. URL: http://tuprints.ulb.tu-darmstadt.de/5937/.

[4] Steven Arzt and Eric Bodden. “StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, 725–735. ISBN: 9781450339001. DOI: 10.1145/2884781.2884816. URL: https://doi.org/10.1145/2884781.2884816.

[5] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Association for Computing Machinery, 2014, 259–269. ISBN: 9781450327848. DOI: 10.1145/2594291.2594299. URL: https://doi.org/10.1145/2594291.2594299.

[6] Gogul Balakrishnan and Thomas Reps. “Analyzing Memory Accesses in x86 Executables”. In: *Compiler Construction*. Ed. by Evelyn Duesterwald. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 5–23. ISBN: 978-3-540-24723-4.

[7] Alexandre Bartel et al. “Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot”. In: *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*. SOAP ’12. Beijing, China: Association for Computing Machinery, 2012, 27–38. ISBN: 9781450314909. DOI: 10.1145/2259051.2259056. URL: https://doi.org/10.1145/2259051.2259056.

[8] Eric 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*. SOAP ’12. Beijing, China: Association for Computing Machinery, 2012, 3–8. ISBN: 9781450314909. DOI: 10.1145/2259051.2259052. URL: https://doi.org/10.1145/2259051.2259052.

[9] Richard Bonett et al. “Discovering Flaws in Security-Focused Static Analysis Tools for Android using Systematic Mutation”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1263–1280. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bonett.

[10] Z. Berkay Celik et al. “Sensitive Information Tracking in Commodity IoT”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1687–1704. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/celik.

[11] Hongyi Chen et al. “Automatic privacy leakage detection for massive android apps via a novel hybrid approach”. In: *2017 IEEE International Conference on Communications (ICC)*. 2017, pp. 1–7.

[12] Jiongyi Chen et al. “IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing”. In: *NDSS*. 2018.

[13] Byungkwon Choi et al. “APPx: An Automated App Acceleration Framework for Low Latency Mobile App”. In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’18. Heraklion, Greece: Association for Computing Machinery, 2018, 27–40. ISBN: 9781450360807. DOI: 10.1145/3281411.3281416. URL: https://doi.org/10.1145/3281411.3281416.

[14] Aske Christensen, Anders Møller, and Michael Schwartzbach. “Precise Analysis of String Expressions”. In: *BRICS Report Series* 10.5 (2003). DOI: 10.7146/brics.v10i5.21776. URL: https://tidsskrift.dk/brics/article/view/21776.

[15] G. Chu, N. Apthorpe, and N. Feamster. “Security and Privacy Analyses of Internet of Things Children’s Toys”. In: *IEEE Internet of Things Journal* 6.1 (2019), pp. 978–985.

[16] Council of European Union. *Regulation (EU) 2016/679*. 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj.

[17] William Enck et al. “TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *ACM Trans. Comput. Syst.* (2014). DOI: 10.1145/2619091. URL: https://doi.org/10.1145/2619091.

[18] Pascal Gadient et al. “Web APIs in Android through the Lens of Security”. In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2020, pp. 13–22. DOI: 10.1109/SANER48275.2020.9054850.

[19] Grant Ho et al. “Smart Locks: Lessons for Securing Commodity Internet of Things Devices”. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’16. Xi’an, China: Association for Computing Machinery, 2016, 461–472. ISBN: 9781450342339. DOI: 10.1145/2897845.2897886. URL: https://doi.org/10.1145/2897845.2897886.

[20] Y. Jia et al. “Burglars’ IoT Paradise: Understanding and Mitigating Security Risks of General Messaging Protocols on IoT Clouds”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 465–481. DOI: 10.1109/SP40000.2020.00051.

[21] Haojian Jin et al. “Why Are They Collecting My Data? Inferring the Purposes of Network Traffic in Mobile Apps”. In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 2.4 (Dec. 2018). DOI: 10.1145/3287051. URL: https://doi.org/10.1145/3287051.

[22] Jeongmin Kim et al. “Enabling Automatic Protocol Behavior Analysis for Android Applications”. In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’16. Irvine, California, USA: Association for Computing Machinery, 2016, 281–295. ISBN: 9781450342926. DOI: 10.1145/2999572.2999596. URL: https://doi.org/10.1145/2999572.2999596.

[23] Deepak Kumar et al. “All Things Considered: An Analysis of IoT Devices on Home Networks”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1169–1185. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-deepak.

[24] William Landi. “Undecidability of Static Analysis”. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: https://doi.org/10.1145/161494.161501.

[25] Ding Li et al. “String Analysis for Java and Android Applications”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, 661–672. ISBN: 9781450336758. DOI: 10.1145/2786805.2786879. URL: https://doi.org/10.1145/2786805.2786879.

[26] L. Li et al. “IccTA: Detecting Inter-Component Privacy Leaks in Android Apps”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 280–291. DOI: 10.1109/ICSE.2015.48.

[27] Li Li et al. “DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, 318–329. ISBN: 9781450343909. DOI: 10.1145/2931037.2931044. URL: https://doi.org/10.1145/2931037.2931044.

[28] Li Li et al. “Static analysis of android apps: A systematic literature review”. In: *Information and Software Technology* 88 (2017), pp. 67 –95. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0950584917302987.

[29] Yilian Li et al. “IoT-APIScanner: Detecting API Unauthorized Access Vulnerabilities of IoT Platform”. In: *2020 29th International Conference on Computer Communications and Networks (ICCCN)*. 2020, pp. 1–5. DOI: 10.1109/ICCCN49398.2020.9209626.

[30] Samin Yaseer Mahmud et al. “Cardpliance: PCI DSS Compliance of Android Applications”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1517–1533. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/mahmud.

[31] D. Mauro Junior et al. “A Study of Vulnerability Analysis of Popular Smart Devices Through Their Companion Apps”. In: *2019 IEEE Security and Privacy Workshops (SPW)*. 2019, pp. 181–186.

[32] Abhinav Mohanty and Meera Sridhar. *HybriDiagnostics: Evaluating Security Issues in Hybrid SmartHome Companion Apps*. Mar. 2021.

[33] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. “Practical Extensions to the IFDS Algorithm”. In: *Compiler Construction*. Ed. by Rajiv Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 124–144. ISBN: 978-3-642-11970-5.

[34] Xiang Pan et al. “FlowCog: Context-aware Semantics Extraction and Analysis of Information Flow Leaks in Android Apps”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1669–1685. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/pan.

[35] Elkana Pariwono et al. “Don’t Throw Me Away: Threats Caused by the Abandoned Internet Resources Used by Android Apps”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ASIACCS ’18. Incheon, Republic of Korea: Association for Computing Machinery, 2018, 147–158. ISBN: 9781450355766. DOI: 10.1145/3196494.3196554. URL: https://doi.org/10.1145/3196494.3196554.

[36] Lina Qiu, Yingying Wang, and Julia Rubin. "Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, 176–186. ISBN: 9781450356992. DOI: 10.1145/3213846.3213873. URL: https://doi.org/10.1145/3213846.3213873.

[37] Marianna Rapoport et al. "Who You Gonna Call? Analyzing Web Requests in Android Applications". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 80–90. DOI: 10.1109/MSR.2017.11.

[38] S. Rasthofer et al. "DroidSearch: A tool for scaling Android app triage to real-world app stores". In: *2015 Science and Information Conference (SAI)*. 2015, pp. 247–256. DOI: 10.1109/SAI.2015.7237151.

[39] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks". In: Jan. 2014. ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23039.

[40] Vaibhav Rastogi et al. "Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android". In: *Security and Privacy in Communication Networks*. Ed. by Bhavani Thuraisingham, XiaoFeng Wang, and Vinod Yegneswaran. Cham: Springer International Publishing, 2015, pp. 256–276. ISBN: 978-3-319-28865-9.

[41] Joel Reardon et al. "50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon.

[42] Jingjing Ren et al. "Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach". In: *Proceedings of the Internet Measurement Conference*. IMC '19. Amsterdam, Netherlands: Association for Computing Machinery, 2019, 267–279. ISBN: 9781450369480. DOI: 10.1145/3355369.3355577. URL: https://doi.org/10.1145/3355369.3355577.

[43] Jingjing Ren et al. "ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic". In: *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys '16. Singapore, Singapore: Association for Computing Machinery, 2016, 361–374. ISBN: 9781450342698. DOI: 10.1145/2906388.2906392. URL: https://doi.org/10.1145/2906388.2906392.

[44] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, 49–61.

ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

[45] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. "Understanding Worldwide Private Information Collection on Android". en. In: *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, 2021. ISBN: 978-1-891562-66-2. DOI: 10.14722/ndss.2021.24076. URL: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_3B-3_24076_paper.pdf (visited on 04/13/2021).

[46] Marcos Tileria, Jorge Blasco, and Guillermo Suarez-Tangil. "WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS". In: *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 63–75. ISBN: 978-1-939133-18-2. URL: https://www.usenix.org/conference/raid2020/presentation/tileria.

[47] Raja Vallee-Rai and Laurie J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. 1998.

[48] Raja Vallée-Rai et al. "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" In: *Compiler Construction*. Ed. by David A. Watt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 18–34. ISBN: 978-3-540-46423-5.

[49] Xueqiang Wang et al. "Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1151–1167. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-xueqiang.

[50] Fengguo Wei et al. "Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, 1329–1341. ISBN: 9781450329576. DOI: 10.1145/2660267.2660357. URL: https://doi.org/10.1145/2660267.2660357.

[51] M. Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

[52] Haohuang Wen et al. "Automated Cross-Platform Reverse Engineering of CAN Bus Commands from Mobile Apps". In: *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*. San Diego, CA, 2020.

[53] N. Wongwiwatchai, P. Pongkham, and K. Sripanidkulchai. "Comprehensive Detection of Vulnerable Personal Information Leaks in Android Applications". In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2020, pp. 121–126.

[54] Daniel Wood, Noah Apthorpe, and Nick Feamster. "Cleartext Data Transmissions in Consumer IoT Medical Devices". In: *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*. IoTS&P '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, 7–12. ISBN: 9781450353960. DOI: 10.1145/3139937.3139939. URL: https://doi.org/10.1145/3139937.3139939.

[55] Q. Zhao et al. "Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1106–1120. DOI: 10.1109/SP40000.2020.00072.

[56] Wei Zhou et al. "Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms". In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. Santa Clara, CA, USA: USENIX Association, 2019, 1133–1150. ISBN: 9781939133069.

[57] C. Zuo, Z. Lin, and Y. Zhang. "Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1296–1310. DOI: 10.1109/SP.2019.00009.

[58] Chaoshun Zuo et al. "Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 1469–1483. ISBN: 9781450367479. DOI: 10.1145/3319535.3354240. URL: https://doi.org/10.1145/3319535.3354240.