Security and Privacy in a World of Interconnected Devices

Marcos Tileria

Information Security Group
Royal Holloway University of London

A thesis submitted in fulfillment of the requirements
for the degree of *Doctor of Philosophy*

February 2023

Dedicated to my parents
This thesis is a small gift to repay your immense sacrifice.
Gracias papá y mamá

Declaration

The work presented in this thesis results from original research conducted by myself in collaboration with others. My contributions and those of the others have been explicitly indicated in the text. I confirm that appropriate credit has been given within this thesis where reference has been made to other works. This research has not been submitted for any other degree or award in any other university or educational establishment.

Marcos Tileria
February 2023

Acknowledgements

I would like to express my deepest appreciation to my supervisor Jorge Blasco. I'm grateful for the extraordinary support and guidance that Jorge provided me throughout this journey. I would like to extend this appreciation to Santanu Dash for the guidance, enriching discussions and the final push to finish this thesis. Their contributions were fundamental for completing this dissertation, from the ideas, recommendations, challenges, and constructive feedback. It was a pleasure working with them.

I would like to thank Fabio Pierazzi and Darren Hurley-Smith for agreeing to act as my PhD committee and for reading this thesis.

I'm grateful to the Centre for Doctoral Training that offered me the opportunity to pursue my studies and provided all the support in terms of courses, seminars, travels, and socials. Thanks should also go to all the academic and administrative staff, the students, and especially all the folks from my cohort. In particular, I must also thank Jordy and Angela for their friendship and for being there during difficult times.

Special thanks to Guillermo Suarez-Tangil for helping me during the first steps of this thesis. Thanks to Helena Gomez for sharing her expertise and helping during the last stage. I want to thank the members of the S3Lab. I had a great experience sharing with all. This work has the footprint of our discussions, seminars, and meetings. I gratefully acknowledge the assistance of Sergio and Guiseppe in helping with the tedious work of labelling datasets.

I very much appreciate the support of my family and friends. Many people helped me across these years with a word of encouragement or by discussing my projects and reviewing manuscripts. Luis, Francisco, Mateo, Nery, Julian, Teddy, Marta, Wilson and the always-present Athena. You are awesome! I'll always be grateful to you.

The most important acknowledgement is for my wife. We started this adventure together, and I'm deeply grateful for your personal support and your help in my work by asking endless questions, reviewing my work, and always being there. Now we see the completion of this adventure and the beginning of many more. Bth, Thanks for everything!

Abstract

In a world of interconnected devices, app-based ecosystems enable a seamless user experience across devices. Although convenient for users, this expanded ecosystem also exacerbates security and privacy threats by exposing users' sensitive data to a broader context. This research looks into multi-platform apps and addresses the question of whether information flows can be detected across different app platforms and how to do it efficiently.

To answer this question, we first instantiate the problem in the wearable ecosystem by analysing platform-specific abstractions and modes of interaction. We identify limitations of current approaches to detect sensitive data transmitted across Mobile-Wear channels and develop a custom static analysis framework that augments the capabilities of taint tracking, enabling inter-device analysis of applications. Our framework enables the detection of information flows that otherwise would remain undetected. Second, we study information flows in the Android TV ecosystem and identify the differences with the mobile ecosystem. In particular, we analyse the behaviour of TV apps in terms of sensitive data collection and communication with other devices using a pipeline of static and dynamic analysis experiments

Analysing these two platforms provided us with valuable lessons to think about arbitrary ecosystems. One common task, for any platform, is generating taint specification for information flow analysis. Therefore, we propose a framework that models the semantics of API methods using Natural Language Processing techniques and software documentation instead of a code base approach. Our framework allows security analysts to detect security-sensitive methods automatically and is robust against software evolution. Thus, our framework is an excellent option for generating taint specifications for arbitrary app platforms.

This investigation contributes to the community by studying two overlooked ecosystems and provides the means to analyse arbitrary app ecosystems. Our methodology is based on a dual-channel perspective: Program Analysis and Natural Language Processing. We use these complementary techniques to better understand Android platforms' security and privacy risks, and we take one step further towards safer ecosystems.

Contents

List of Figures

List of Tables

1 Introduction

This chapter contains the problem statement, the motivations for this thesis, our contributions, and the published works, including the open-source tools we provide to the community.

The rapid development of smartphones and Internet of Things (IoT) devices enables an app-based ecosystem where users can interact with the digital world from any device. For instance, wearable devices can now run apps on appliances with large computing, storage, and networking capabilities. Similarly, Smart TV vendors have expanded their user base due to the popularity of streaming services, affordable prices, and enhanced user experience [1].

A key feature of these devices is that they are interconnected and provide a usable interface to interact with smart devices and cloud-based apps. For instance, users can make payments or monitor their health with gadgets seamlessly from any device such as smartphones, smartwatches or personal assistants. Furthermore, users can control devices in their smart homes using proximity protocols or sending remote commands over the Internet [2–4]. In general, smart devices are in a rich environment where they can interact with other devices. A 2022 survey [5] reported that US households have an average of 22 connected devices, highlighting the increasing prevalence and integration of smart technology into our daily lives.

Interconnected ecosystems offer a myriad of additional functionalities and enhanced user experience. On the other hand, these improvements also introduce vulnerabilities and potential privacy violations. Several studies have exposed vulnerabilities in smartwatches and their ecosystem [6–9], including their Bluetooth connectivity [8,10] and privacy breaches [11,12]. Previous works reported vulnerabilities and data leakage in IoT devices, including Smart TVs and Chromecast devices [2,4,13,14], while another line of research focused on detecting vulnerabilities and attacks on smart home platforms [2–4,15,16].

Many companies rely on analytics and advertising as part of their business model. Application developers embed third-party libraries (TPL) in their apps for these purposes. TPLs are tightly integrated with the host app and share the same execution context. This architecture exposes sensitive data to third parties. Several works analysed privacy implications in mobile apps [17–21] and the tracking and advertising ecosystem [17,18,22–24]. The mobile platform does not stand alone with privacy problems. The TV vendor Vizio paid a large fine for collecting viewing data on 11 million consumers without their knowledge or consent [25]. While other works reported several vulnerabilities on Smart TVs [2,13,14,26,27], including personal data leaks in Roku and Fire TV [28,29].

The problems raised above demonstrate the prevalence of security and privacy problems in app-

based ecosystems. Moreover, studies have shown that people are concerned about the security of smart devices and privacy expectations of their personal data [5,12,30]. With this context in mind, *this thesis looks at multi-platform apps and how sensitive information is used. Specifically, we look into information flows across multiple apps from different platforms.*

Each platform consists of different devices designed to provide a custom user experience and deployed in different environments. For instance, wearable devices are designed to provide a wrist experience focusing on critical tasks and short interactions. In contrast, smartphones are designed for a complete experience and prolonged interactions. Different devices and execution environments make the analysis of each platform cumbersome. However, apps are the most common way to interact with interconnected ecosystems, and we capitalise on this scenario. Thus, we choose to study apps customised for each platform to simplify the analyses instead of relying on complex testing environments. Note that we study third-party applications that can be downloaded from the official or alternative stores, but software from IoT/medical devices are outside the scope of this thesis.

Our decision to target custom apps leads to the problem of choosing a target platform. Android, Apple, and Amazon are among the most popular vendors that offer numerous smart devices. We choose Android as the target platform for multiple reasons: 1) Android's open-source nature facilitates the analysis of the Android framework and apps. 2) Its availability across devices, i.e., phones, TVs, watches and cars. 3) Its high integration across devices facilitates consumer adoption, amplifying the affected user base. 4) Its predominance in the mobile environment. Android is the most popular OS, with over 3 billion active devices worldwide. 5) The Android research community is very active and has laid the groundwork for other researchers to study more specific problems.

1.1 The Problem

Apps across platforms generally implement similar core abstractions while including specific features. We are interested in understanding if current information flow techniques generalise to capture these specific features and the limitations arising from platform constraints. We are also interested in uncovering platform-specific problems, which might not necessarily be related to information flows but have an impact on our ability to understand threats to users' privacy. Thus, we study security and privacy threats in arbitrary Android platforms targeting their apps.

Over the years, Android has evolved to support a wide range of platforms to provide a multi-device experience for its users. While smartphones are still the most popular device, the market is constantly diversifying with additional devices, e.g., Smart Cars. Google maintains a customised Operating System (OS) version for each Android platform considering hardware limitations and user experience. This diversification compels developers to create or adapt their apps for each platform. In this thesis, we refer to a platform as a specific instance of one ecosystem. For instance, the Android TV platform is the Android instance of the Smart TV ecosystem.

Figure 1.1 shows multiple versions of a hypothetical Android app. The figure shows this app's

multiple versions: wearable, TV, mobile and car. The app's main function is the same across all platforms. However, significant differences arise when looking at their technical details. For instance, mobile and wearable apps can interact in close proximity using platform-specific APIs, while the TV app receives inputs from a remote control. These platform-specific capabilities suggest that threats vary across platforms. Therefore, the methods for identifying these threats also differ. As app developers must adapt or create apps from scratch for new platforms, researchers may need to do the same with their framework or testing environment. We highlight the importance of a better understanding of these differences and current techniques' limitations to model app behaviour across platforms.

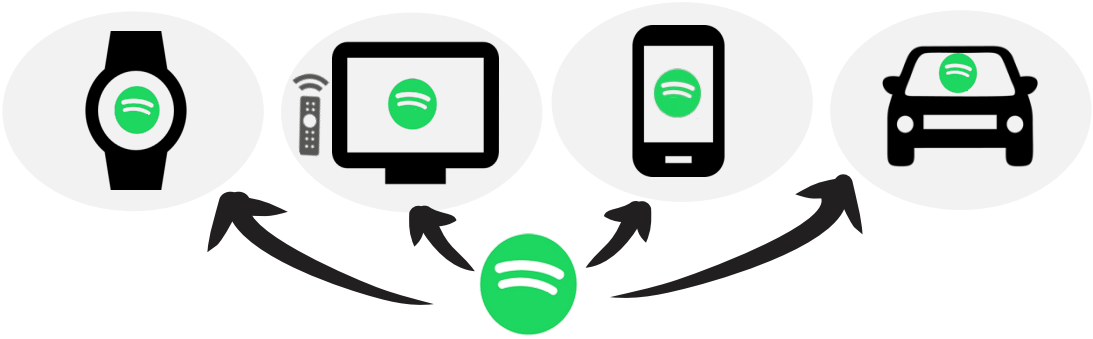


Figure 1.1: Multi-platform app example

The Android research community has made a great effort to study Android apps to confront potential vulnerabilities and threats to the growing ecosystem [31]. These works use static analysis [32–37, 37–44] and dynamic analysis [15, 45–49] to uncover privacy violations and vulnerabilities. However, most of these works focus on mobile apps only, and threats on other platforms are overlooked. In this thesis, we seek to fill this gap by expanding the scope from the mobile ecosystem to an ecosystem of interconnected devices focusing on Android apps. It is important to mention that our research effort centres around studying these ecosystems from a static analysis perspective, even though we complement our evaluation with dynamic analysis in some cases. We present the `Research Questions` and describe our methodology below.

1.2 Research Questions and Contributions

Our research aims to improve the security and privacy of Android apps across different platforms. While most research efforts have been devoted towards mobile apps, less attention is given to apps for other smart devices and how they interact with their mobile counterparts. This thesis focuses on making existing analyses aware of the differences between apps across platforms. With this in mind, we propose the following research questions, and we associate these questions with specific chapters, contributions, published works, and open-source tools.

RQ1. ***Can sensitive information flows be detected in arbitrary app platforms?***
This thesis centres around answering this question. To address the question above, we first examine the similarities and differences between multiple Android platforms. Then, we present two case studies where we analyse abstractions and interactions that are platform-

specific. In particular, we study sensitive information flows in Wear OS and Android TV apps and compare them with the mobile platform when possible. We seek to identify the limitations of state-of-the-art frameworks to detect sensitive information flows in arbitrary platforms and propose solutions to fill the gaps.

RQ2. ***Do other platforms present problems unseen in the mobile platforms?***
This question attempts to uncover problems that still need to be revealed to the community. Assume that it is possible to detect information flows in arbitrary platforms. What other problems are we missing because of the lack of attention to platforms such as Wear OS and Android TV? In this thesis, we examine these platforms to uncover problems related to modes of interaction, permissions, third-party libraries, and metadata. We aim to contribute to a much better understanding of app-based ecosystems and the divergences from Android mobile.

RQ3. ***Is it possible to automate the analysis of arbitrary platforms?***
This question focuses on the practicalities of analysing arbitrary app ecosystems. Assume again that it is possible to detect information flows in arbitrary platforms. What are the difficulties that security analysts face when analysing different platforms? In this thesis, we explore the idea of automating the generation of taint specifications from software documentation. Automation aims to scale the analysis of real-world apps and reduce the burden of manual analysis of apps, the Android Framework and Google libraries.

If existing techniques are enough to detect information flows in arbitrary apps, then the answer to RQ1 is yes, and we only need to focus on platform-specific problems (RQ2) and automating the evaluation of multi-platform apps (RQ3). However, we show that existing techniques lack the capabilities to capture abstraction from all the platforms, preventing the detection of sensitive information flows in many scenarios. The figure 1.2 illustrates our methodology to address these questions. Our approach consists of two phases: `Platforms evaluation` and `Generalisation`.

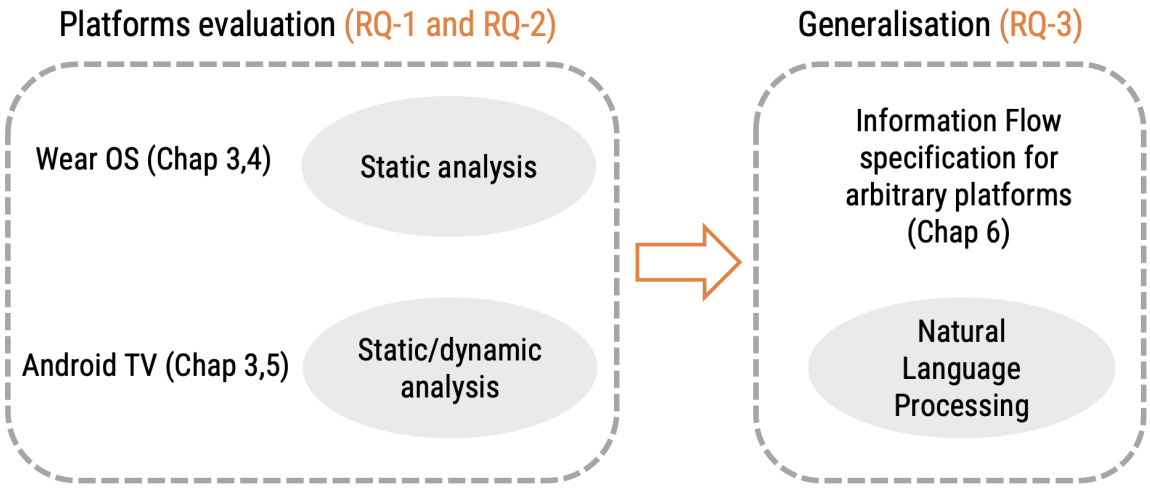


Figure 1.2: Thesis methodology

In the first phase, we evaluate two Android platforms (Wear OS and Android TV). First, we present structural components of the Android framework and Google Play Service libraries. Then, we use static and dynamic analysis techniques to study the structure of Android apps, sensitive APIs modes of interaction, and information flows potentially leaking sensitive data within and across platforms. We evaluate our approach using custom benchmarks and large data sets of real-world apps. This phase covers the RQ1 and RQ2.

In the second phase, we use the lessons learned from these evaluations to propose methods to automate critical parts of the security evaluation of multi-platform apps (RQ3). In particular, we design and implement a framework that uses Large Language Models [50] to automatically generate information flow specifications for arbitrary Android platforms. These specifications are then used as input for information flow analysis frameworks. This approach produces precise specifications without the need to study the platform architecture or available APIs.

Our work leads to practical ways to examine multi-platform apps. In our experience, scanning apps from new platforms involves a lot of manual work. Therefore, we design and develop solutions that scale to analyse real-world apps from arbitrary platforms. During this investigation, we encounter many challenges that we outline below[1].

Custom Analysis per Platform. Android apps are developed considering the resources available on each specific platform. These resources refer to API methods, libraries, permissions, and third-party libraries. Therefore, apps from different platforms differ despite using the same programming language and architecture. This makes existing static and dynamic analysis frameworks, developed for mobile apps, not well suited for other platforms in many cases.

Software Evolution. The Android framework rapidly evolves to keep up with new hardware, functionalities and user experience. Android has 33 API levels and 13 major versions as of January 2023. New classes and methods are introduced and removed on each release. Developers and researchers must cope with these changes to keep their work relevant. It has been shown that both have problems keeping the same pace [51–53]. Considering the constant API changes, one challenge is generating security specifications for information flow analysis for different platforms and releases.

Closed Components. Android is an open-source OS, but not all building blocks are open-source. For instance, Android apps heavily rely on Google libraries (more than 30) to provide services such as maps and payment [23,52,54]. These proprietary libraries reduce static analysis coverage as a big chunk of the code is unavailable.

Lack of app benchmarks. While there are multiple mobile app benchmarks to test static analysis frameworks [32, 35], other platforms lack similar benchmarks making it difficult to evaluate different tools.

This study aims to answer the Research Questions considering the challenges stated above.

[1]Challenges specific to information flows in Android apps are described in Section 2.3.3

Overall, the main contributions of this thesis are summarised as follows:

- **Platform characterisation**. We first identify the similarities and differences of multi-platform Android apps. With this information, we detect common threats and problems that are platform-specific (RQ1, RQ2).
- **Wear OS platform analysis**. We study information flows generated by the interaction of mobile and wearable apps. For this, we develop a static analysis framework, `WearFlow`, that enables the analysis of sensitive information flows transmitted through Mobile-Wear channels (RQ1, RQ2). We show that `WearFlow` detects information flows bypassed by current approaches and scales to detect privacy violations in real-world apps, including obfuscated apps.
- **Benchmark suite for Wear OS**. We develop the first app benchmark `WearBench` to analyse mobile and wearable apps interactions. The benchmark consists of 15 pair of Mobile-Wear Android apps that contains all APIs that can be used to share information between mobile app and the wearable companion apps (RQ2). We evaluate `WearFlow` using `WearBench` apps.
- **Android TV platform analysis**. We present a deep analysis of the Android TV ecosystem using a dataset of more 4.5K TV apps. We seek to detect privacy violations and identify platform-specific problems (RQ1, RQ2). For this, we study information flows using a pipeline of static analyses complemented with traffic analysis experiments. We found a prevalence of static identifiers for tracking purposes despite this not being the recommendation, limiting the effectiveness of Google's privacy policies. Moreover, we found many bad development practices, some of which are specific to the Android TV ecosystem.
- **Automatic security specifications from software documentation**. We propose a novel approach to generate taint specification for static information flow analysis of Android apps for arbitrary platforms (RQ3). `DocFlow` is a framework that models the semantics of API methods using their documentation to detect sensitive methods and assigns them semantic labels. Security analyst can use our framework to reduce the manual effort of analysing code and documentation to generate security specifications. `DocFlow` achieves better performance than baseline approaches and does not require access to app or framework code, instead it relies on publicly available documentation and state-of-the-art Natural Language Processing techniques.

1.3 Thesis Overview

Chapter 2 introduces the background information required to follow the rest of the thesis. In particular, we describe the Android architecture, app components, and security model. Then, we focus on the program analysis techniques we use in this thesis. Finally, we provide the preliminaries to understand distributed word representation in Natural Language Processing.

Chapter 3 describes the different Android platforms and introduces the Google Play Services

architecture. These libraries will be used to model Mobile-Wear communications and generate security specifications for arbitrary platforms. Additionally, we analyse similarities of Wear OS Android TV apps with mobile apps and their differences. This chapter serves as a prelude to answer questions RQ1 and RQ2 and to understand the threats for wearable and TV apps.

Chapter 4 examines the Mobile-Wear interaction and presents WearFlow, our static analysis tool to detect sensitive information flows across mobile and wearable apps. We describe WearFlow internals, including its module to deobfuscate APKs, the new app benchmark `WearBench` to analyse mobile-wear communications, and our experiments using a real-world apps dataset. This chapter exposes the limitation of current approaches to deal with wearable apps (RQ1).

Chapter 5 presents the analysis of the Android TV ecosystem. Here, we analyse TV apps in terms of sensitive data collection, communication capabilities, and the prevalence of tracking and advertising libraries. Additionally, this chapter also compares popular TV apps with their mobile counterpart and show that some problems are unique of the TV ecosystem. This chapter uncovers problems that are specific to the TV ecosystem (RQ2).

Chapter 6 presents `DocFlow`, a framework that uses NLP to capture API methods semantics directly from the documentation. `DocFlow` can be used to detect sensitive methods, classify methods into semantic categories, and it is able to find semantic-similar methods across Android versions. `DocFlow` can be used to generate taint analysis specifications and extract semantic properties from software documentation for arbitrary platforms (RQ3).

Last, chapter 6 concludes this thesis by discussing our results and portraying the key findings in the Android ecosystem. Table 1.1 shows the mapping of each Research Question with their corresponding chapters. Chapters 1*, 2* and 7* correspond to the introduction, background, and conclusions.

	Chapters						
Research Questions	1*	2*	3	4	5	6	7*
RQ1			**x**	**x**			
RQ2			**x**		**x**		
RQ3						**x**	

Table 1.1: Chapters and Research Questions

1.4 Published works and Artefacts

This thesis is based on three papers which are the product of my work with the guidance of my supervisors Jorge Blasco, Santanu Dash, the collaboration of Guillermo Suárez-Tangil and others who provided valuable input and feedback. They are properly acknowledged at the beginning of the thesis. All the experiments were conducted by myself.

1. Tileria, M., Blasco, J. and Suarez-Tangil, G., 2020. WearFlow: Expanding Information Flow Analysis To Companion Apps in Wear OS. In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020) (pp. 63-75). The content of

this paper is discussed in chapter 3, and chapter 4 in more details (Related to RQ1 and RQ2).

2. Tileria, M. and Blasco, J., 2022. Watch Over Your TV: A Security and Privacy Analysis of the Android TV Ecosystem. Proceedings on Privacy Enhancing Technologies, 3, pp.692-710. The content of this paper is discussed in chapter 3, and chapter 5. Correspond to the Research Questions RQ1 and RQ2.

3. DocFlow: Extracting Taint Specifications from Software Documentation. Manuscript under revision. This work is discussed in chapter 6, and relates to the Research Question RQ3.

Finally, we make available the tools, benchmark, and dataset we used in this research, so that the community can benefit for further analysis or reproduce our experiments. These can be downloaded from the repositories listed below.

1. `WearFlow` repository (RQ1,RQ2). https://gitlab.com/s3lab-rhul/android/wearflow

2. `WearBench` repository (RQ1,RQ2). https://gitlab.com/s3lab-rhul/wearbench

3. Android TV repository (RQ1,RQ2). https://gitlab.com/s3lab-rhul/watch-over-your-tv-paper

4. `DocFlow` repository (RQ3). https://gitlab.com/s3lab-rhul/android/docflow

My research was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

2 Background

This chapter contains the background information to support the remaining chapters of this thesis. We start by dissecting the Android architecture and application components. Then, we describe the program analysis techniques we use to analyse Android apps. Finally, we give the basics to understand distributed representation of documents in Natural Language Processing.

Android has been expanding its platform since 2008 from a mobile OS to a myriad of platforms such as Wear OS, Android Auto, Android of Things[1], and Android TV which compose the Android ecosystem. We describe these platforms in the next chapter, including the commonalities and differences across platforms, and focus this chapter on applications and their representation. Overall, apps provide an interface to the digital world for each Android platform, and we study them to understand how threats in the mobile platform portrait in other platforms. Next, we describe the Android architecture and app structure. With this background information, we aim to expose the scope of this thesis and present the fundamentals to understand its main contributions, particularly those of chapters 4, 5, and 6.

2.1 Android Architecture

Android is an open-source operating system (OS) initially designed for mobile devices but later expanded to other smart platforms. The Android Open Source Project (AOSP) repository contains the OS source code, information to create Android variants, and API documentation. Over time, Android decoupled many critical system components in closed-source libraries and a system-level APK known as Google Play Services. The Android system architecture is divided into multiple layers. Figure 2.1 shows an overview of the Android system architecture.

- The **Application Framework**, also known as the Java API Layer, contains the Java APIs that provide access to the building blocks to create Android apps. This layer simplifies the development and facilitates the reuse of system components and services. Our Android TV and Wear OS analyses focus on the Application Framework.

- The Binder Inter-Process Communication (**Binder IPC**) is a mechanism that allows the interaction between the Application Framework with the System Services layer. Binder provides a high-level abstraction on top of traditional OS services that facilitates binding resources from one execution environment to another.

- The **System Services** layer is the bridge between the functionalities exposed by the Java APIs and the underlying hardware. These services include window management,

[1] Android of Things was deprecated in 2019 and shut down on January 2022

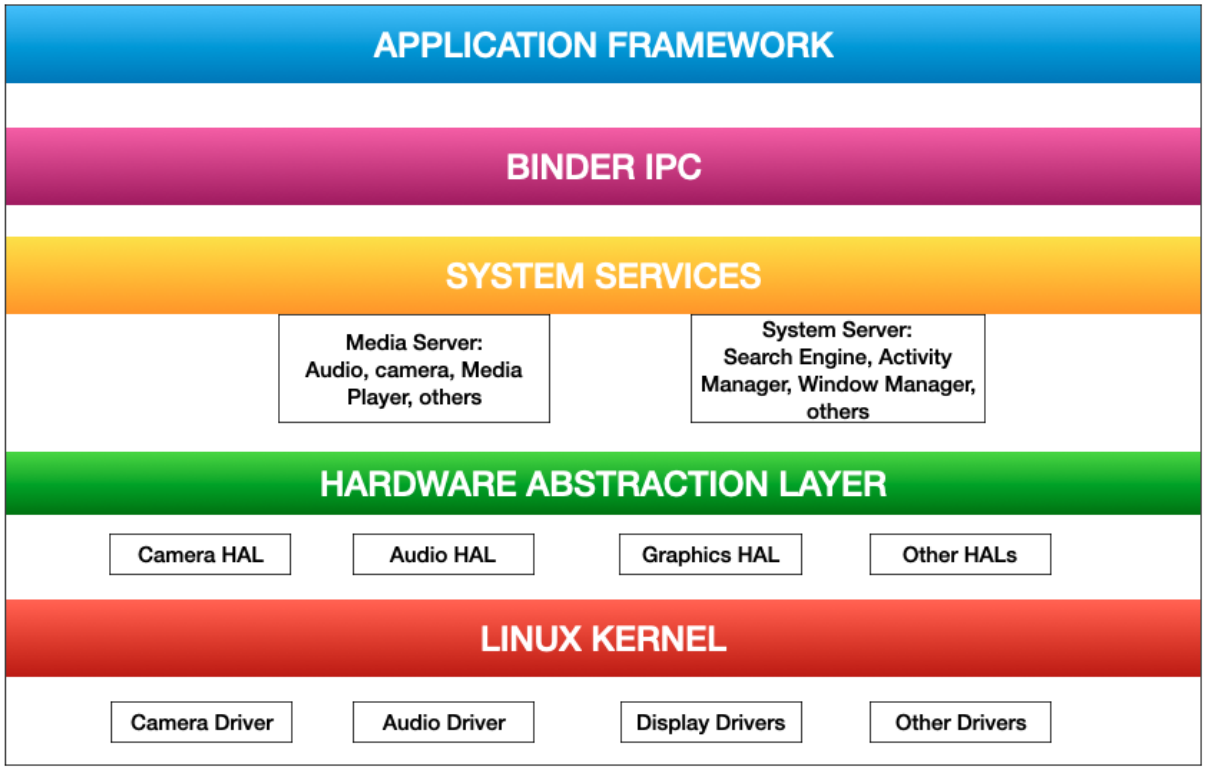


Figure 2.1: Android system architecture.

notifications, searching, and media services.

- **Hardware abstraction layer (HAL)** defines a standard interface for hardware vendors to integrate their hardware with Android. This layer enables Android to be transparent to different driver implementations.

- The **Linux kernel** is a version of the Linux kernel with custom additions such as the memory killer, power management service, the Binder IPC driver, and other features essential for embedded platforms.

2.1.1 Android Apps

An Android app is a software that run on a device that supports the Android OS. These apps are developed for specific purposes such as email, contact management, games, and others. As we observed in Figure 1.1, apps can be adapted to specific platforms. Developers usually publish their apps on well-known markets such as the Play Store [55], where the users can download the version compatible with their smart devices.

Android apps are developed mainly using Java, Kotlin, and native code (C/C++) to a lesser extent. The final code is compiled into Dalvik Executable (DEX) format. DEX files are optimised for performance and packaged into compressed APK files that contain the app's compiled code and resources required to run the application. The Android app Bundle is a publishing format that aims to reduce space in the user's device by separating device-dependent parts. The bundle format is gradually replacing the APK format, which is still the preferred publishing

format in alternative markets. While the Bundle format optimises the shipping and installation process, users still install APK files in their devices. Therefore, we dissect the APK format below. Figure 2.2 shows the structure of APK files.

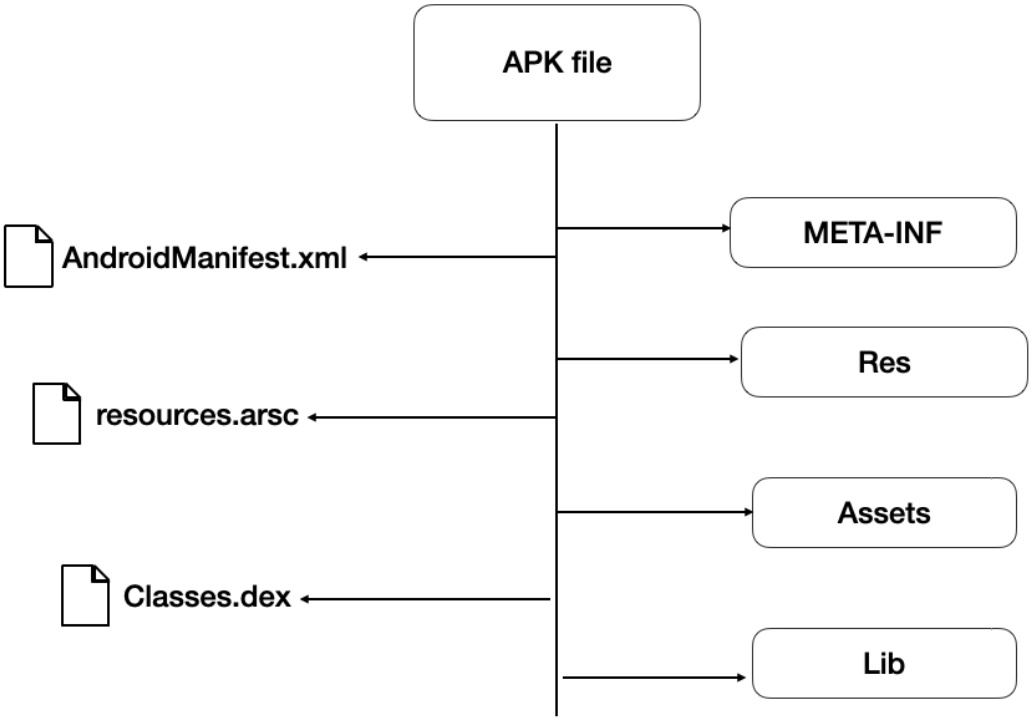


Figure 2.2: APK structure.

The **Android Manifest** is a configuration file that contains general information about the application. For instance, it contains the list of permissions, components, package name, version, and required hardware, among other information. The **META-INF** folder is essentially a manifest of metadata information, including the developer certificate.

DEX files contain Java/Kotlin compiled code into DEX format, the proprietary Google format for DVM. The **Lib** folder stores all the native libraries for different architectures such as x86_64, ARMv8, and ARMv7. Since Android supports different architectures, the presence of a sub-folder indicates the support for that platform. For instance, the arm64-v7a indicates support for ARM7 binaries.

The **Assets** directory contains raw resources (videos, document templates, JavaScript code, or HTML files) required at running time. The **Res** directory stores resources linked at compile time and stored in the file **resources.arsc**. This file links the resources in the **Res** directory to the code in the .DEX files.

Android Runtime. Android applications require a runtime environment to be executed. The Dalvik Virtual Machine (DVM) was the first Android runtime. DVM uses a Just-in-Time (JIT) compiler to convert compiled java programs into .dex and .odex files (optimised dex). Dalvik uses a register-based architecture instead of the stack-based architecture used by the Java Virtual

Machine. ART (Android Runtime) is the new runtime for newer versions (default from Android 6). ART uses a mix of Ahead-of-Time and JIT compilation with improved memory allocation and garbage collection. For this runtime, .dex files are converted to the Executable and Linkable Format (ELF) using the `dex2oat` tool. ART and Dalvik are compatible runtimes environments. Both execute Dalvik bytecode, so apps developed for Dalvik can be executed by ART.

2.1.2 App Components

There are four component types: Activities, Services, Broadcast Receivers, and Content Providers. Each component has a distinct purpose, and these are briefly outline below.

- **Activities** represent a single screen with a user interface, such as writing an email or watching a video. Activities are independent units, although they can work together to present a continuous user experience.
- **Services** allow apps to keep running in the background to perform long tasks or wait for remote processes. Examples include data synchronization tasks at night or playing music with the app running in the background.
- **Broadcast Receivers** listen to events outside of a regular user flow. This allows apps to respond to system-wide events such as network disconnection or low battery. Broadcast Receivers can listen to events generated by other apps and usually acts as a gateway to other component types, e.g., the component receives a system broadcast and pass the control flow to an activity.
- **Content Providers** are components that offer a persistent storage abstraction. Other apps can access data stored in Content Providers if they hold the corresponding permission.

Android apps differ from traditional Java programs in several ways. First, apps do not have a main method or unique entry point. The OS can start one app by invoking any component declared in the Android Manifest file. Second, each component has a **lifecycle** defined by the Android Framework. This lifecycle indicates the default program flow, e.g., Figure 2.3 shows the lifecycle for an Activity. Lifecycle methods are called by either the operating system or framework code depending on the events and the user interaction. Developers can override the lifecycle methods to customise a component flow or define new **callbacks** to handle user interactions, e.g., callbacks to capture click events. These asynchronous events present a challenge when constructing the control flow abstraction because events can appear in arbitrary order. We describe how static analysis tools deal with this problem in Section 2.3.3.

Inter-Component Communication (ICC). Android communication mechanism allows the interaction and data exchange between components. This interaction is achieved by sending **Intents** or using Unified Resource Identifiers (**URIs**) to access data (e.g., Content Providers). Intents are asynchronous messages managed by the Binder IPC. **Intent Filters** are specifications declared in the AndroidManifest that define the capabilities of each component, e.g., the types of messages a component can receive. ICC can occur in the context of one app or across different

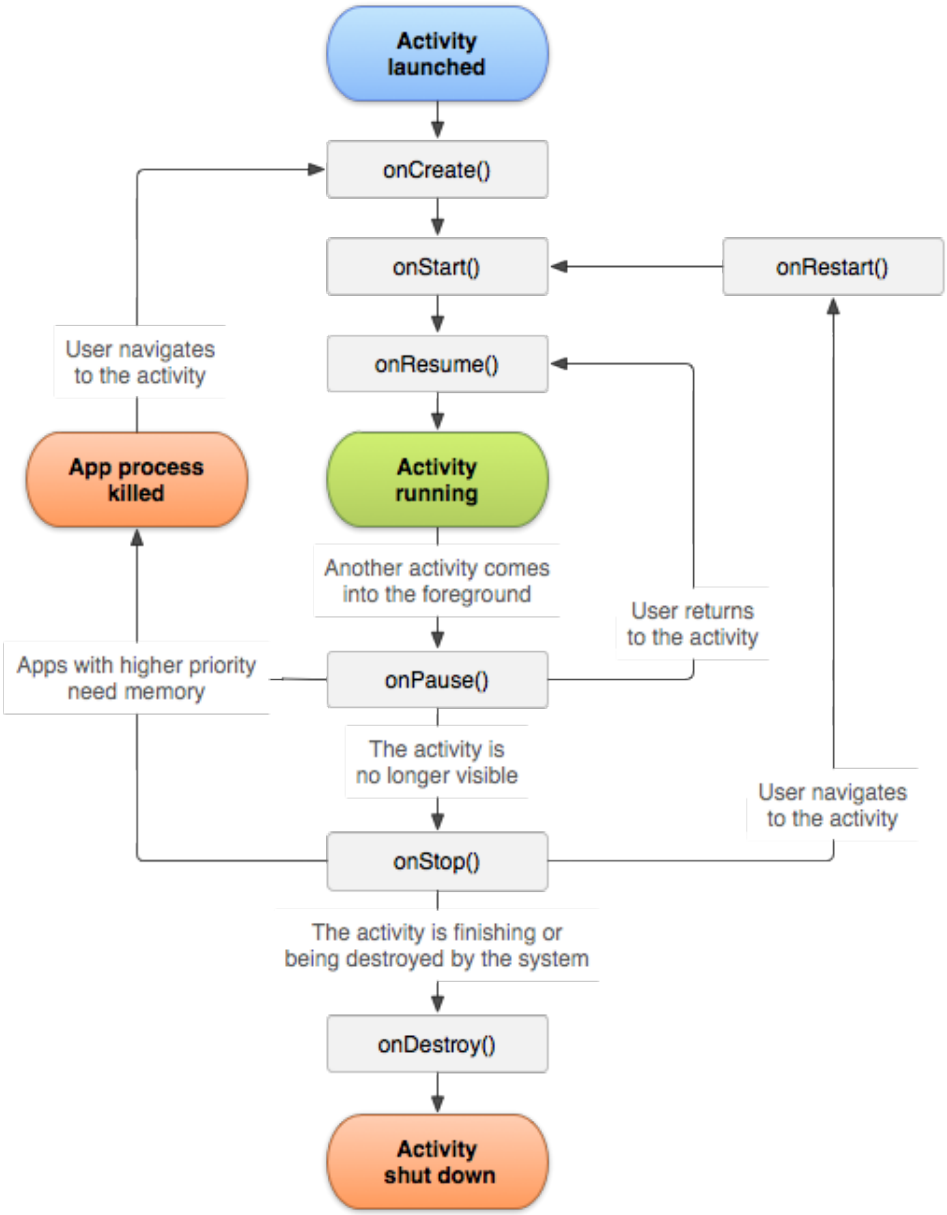


Figure 2.3: Activity lifecycle (adapted from [56])

apps (Inter-App Communication). Additionally, apps can use inter-device communication APIs to connect with other apps on different devices. Some examples include traditional sockets, Bluetooth, and other proximity communication protocols. As we are interested in studying information flows across platforms, we expand on this topic in chapters 3, 4, and 5.

2.2 Apps Security Model

The Android platform has around 3 billion active devices [57]. Securing such an extensive app ecosystem requires industry-leading security features and considering all parties involved. These parties are the Android OS, apps developers, and final users. We give a brief description of them and then focus on the features that are more relevant to our thesis: sandboxing, permission and certificates.

The Android OS security model is built on top of the security elements offered by the Linux kernel. Each layer (Figure 2.1) adds security features to protect its resources. For app developers, Android offers many security features such as security updates through Google Play Services, testing tools, app integrity protection through certificates, and flexible security controls. The final users control the permission requests and have multiple authentication options available. Android also provides security programs and blogs for developers and users.

Sandboxing. Android apps are designed to be secure by default. Each app runs on a sandbox that isolates the app's resources from other apps on the same device. The OS assigns a unique user ID (UID) to each app and sets up a kernel-level sandbox (process and file system) to achieve isolation. Figure 2.4 shows an example of two Android apps isolated through the kernel sandboxing mechanism. In this example, data written by the Spotify app can only be accessed by this app. Other apps, such as Slack, need to request explicit permission to access Spotify resources. Android apps can leak sensitive data by exposing private information outside the sandbox. In the next section, we describe how apps interact with the system through Android and Google Play Services APIs. Section 2.3 describes the technique to track data flows that enter and leave the app's context (sandbox).

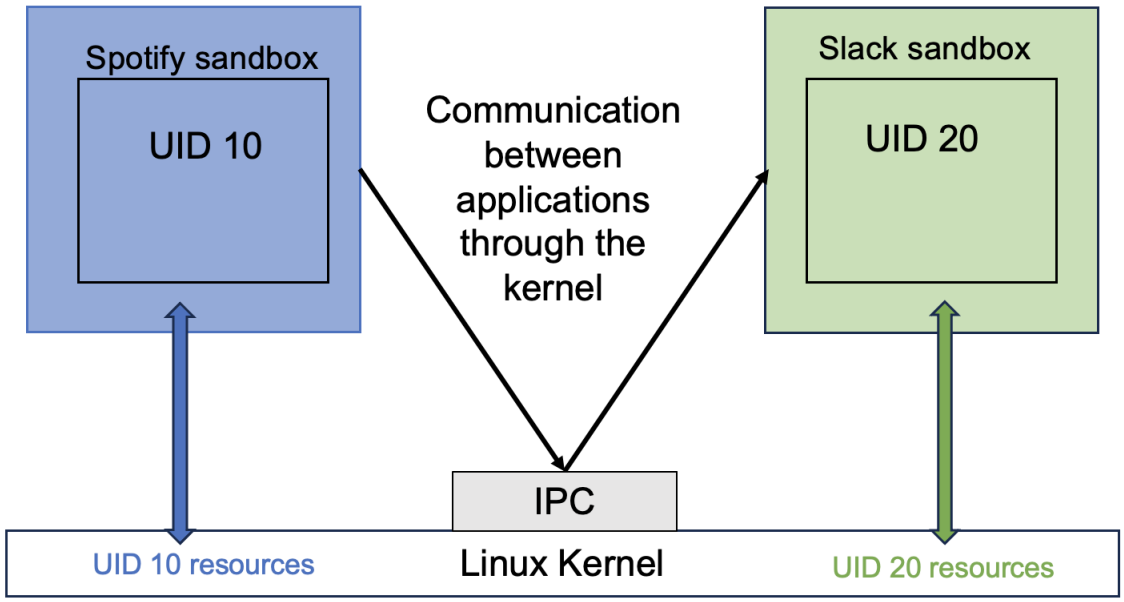


Figure 2.4: Application sandboxing mechanism. Applications can only communicate via IPC (Inter-process communication) through the kernel. Direct communication is not allowed.

Permissions. Android permission mechanism controls access rights to shared resources. These resources are not limited to hardware or features (camera, network access) but also refer to sensitive data such as device identifiers and serial numbers. Apps need to request permission to access shared resources that are located outside their sandbox. For instance, the camera is an external resource protected by the permission `android.permission.CAMERA`. Similarly, the permission `android.permission.LOCATION` protects access to location services. Figure 2.5 shows a permission request example.

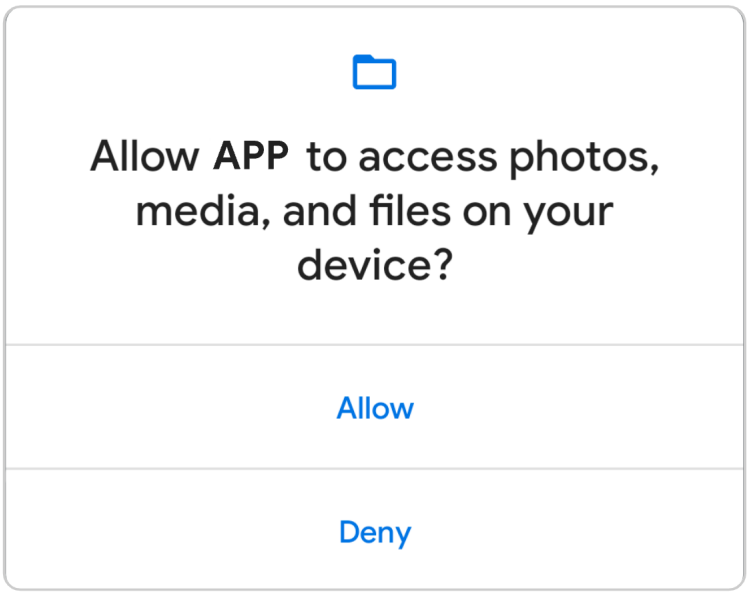


Figure 2.5: Permission request example

There are broadly three permission levels: normal, signature, and dangerous. Normal permissions protect low-risk resources. Signature permissions are for applications with the same certificate. The Android system automatically grants normal and signature permissions at install time (install-time permission). Dangerous permissions protect the most sensitive resources and are granted/revoked by the user at runtime since Android version 6.0. Users can also revoke install-time permissions at running time in newer Android versions. In addition to these three permission levels, third-party apps can define custom permissions to protect or share their resources with other apps. All permission types are declared in the app's Manifest file. Developers must specify a definition for new custom permission, e.g., name and protection level. We discuss the permission model of Wear OS application in Chapter 4 and analyse TV app permissions in a large-scale study in Chapter 5.

App certificates. Android uses a code signing mechanism to ensure that developers consent to actions on their apps [58]. This mechanism prevents malicious apps from injecting/removing code into other apps. Thus, all developers must sign their apps before publishing them in the Play Store. The code signing process generates the UID that the OS uses to create the sandbox for each app (Figure 2.4). Therefore, if a developer uses the same certificate for two or more apps, these apps are allowed to run in the same sandbox. Certificates contain a pair of private/public keys and other metadata that provide information about the private key owner. We use certificate information (fingerprints and metadata) when analysing the Android TV ecosystem (Chapter 4). One important factor to consider is that Android certificates are self-signed. We discuss the implications of trusting self-signed certificates in the same chapter.

2.2.1 Android APIs and Sensitive Data

Android apps are tightly coupled with the runtime environment. This means that apps do not contain the necessary code to run as standalone programs, and they rely on the Application

Framework to access the resources from the lower layers through Java APIs. Thus, the development is heavily based on the API pattern where developers invoke API methods and receive return values, all without requiring the implementation details from the source code. Additional components are normally available via *Google Play Services* libraries (details explained in Chapter 3). These bundle libraries expose stub methods which are implemented in the proprietary app Google Play Services (same name as the library) and work in the same way as the Android Application Framework. We leverage this implementation pattern to generate static taint analysis specifications based on the documentation of these APIs (Chapter 6).

One of the main goals of this thesis is to understand how sensitive data is collected across different platforms. Thus, we first look at how apps access sensitive data in Android. The sandboxing mechanism prevents apps from accessing shared resources. A `shared resource` is an abstraction from the Java API layer that provides access to system services. As we mentioned above, app developers rely on API calls to the Java Framework to access such resources (e.g., location, device identifiers, sensors).

We are interested in data flows that interact with resources outside the app's context, not data flows that simply remain within the boundaries. Consider the code snippet in Listing 2.1 that provides an example of Java APIs' mode of interaction. This example shows a code reading the last known location and then exposing it to other parties through file-system and network-connection APIs. The method `getLastKnownLocation()` in line 2 reads data from the `LocationManager`. The `LocationManager` class is a shared resource that provides access the system location services. In line 7, the method `writeFile` writes the location to the file system, and the method `post` (line 10) writes the location to an HTTP resource.

```
1  public void onCreate(Url url, String filePath){
2      location = LocationManager.getLastKnownLocation()  // source
3      ...
4      //some code
5      ...
6      FileOutputStream file = new FileOutputStream("file_name")
7      file.writeFile(filePath,location). //sink 1
8
9      HttpURLConnection connection = new HttpURLConnection(url)
10     connection.post(url, location)  //sink 2
11     }
```

Listing 2.1: Code reading sensitive data through Android API (sources) and exposing to shared resources (sinks)

Informally, **sources** are methods that read sensitive data, and **sinks** are methods that might leak or expose data. In our previous example, the method `getLastKnownLocation` is a source, and the methods `writeFile` and `post` are sinks. Following the definition used by Rasthofer *et al.* [51], sources are methods that read from a shared resource and return non-constant values into the application code. Likewise, sinks are methods that write a non-constant value to a

shared resource. Security practitioners often need to produce static analysis specifications that include a list of sources and sinks. In particular, detecting potential interactions between apps and shared resources requires identifying API methods that enable these interactions. We refer to these methods as **security-sensitive** methods in this thesis.

The idea of what is sensitive depends on the context of the analysis. For instance, an analyst might only be interested in detecting PII leaked through network requests. In our case, we are interested in detecting potential data leaks, considering any Android device and the interaction across many devices. Thus, we are interested in data flows across applications and not only data leaks to untrusted/malicious parties. These applications might be from the same or a different platform, e.g., mobile with wearable applications.

One important factor to consider when analysing security-sensitive methods is the evolution of the Android ecosystem. Android receives constant updates to cope with new features and hardware. There are 13 versions and 33 API levels, as of February 2023. While a new Android version is associated with many features that are visible to users, API releases are related to changes in the Android framework. Each release contains hundreds of thousands of methods. Previous research shows evidence that app developers usually struggle to keep up with these updates due to their volume [59,60]. On top of API levels, updates from 40 closed-source Google Play Services libraries make the ecosystem even more complex. For instance, Figure 2.6 shows the number of Google Play Services updates per year since 2015.

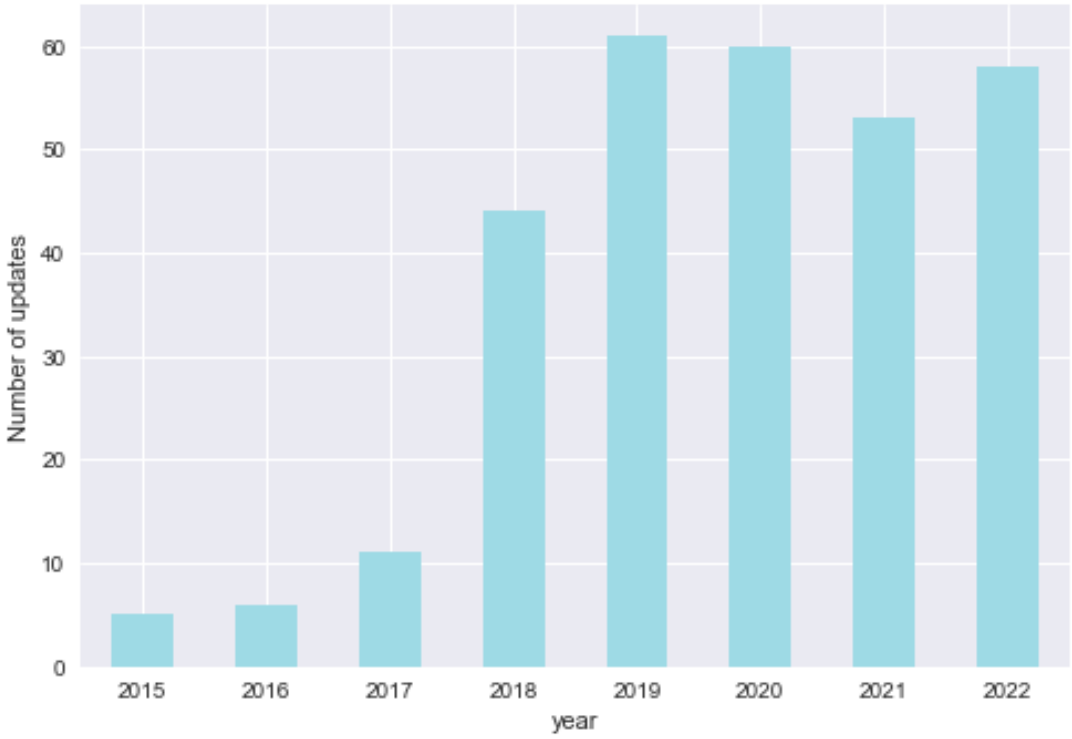


Figure 2.6: Google Play Services updates until November 2022

With the ever-changing Android framework and Google libraries, collecting a comprehensive list of security-sensitive methods becomes critical for analysis completeness. It has been shown that

static analysis tools miss a large number of data leaks due to incomplete configurations [51,61]. In the code example above, if only the network sinks are included in the taint specification, any leak through the file system will remain undetected. Moreover, apps tend to have more sensitive data leaks with newer Android releases [60], augmenting the importance of complete taint specifications. Therefore, it is desirable to automate this task due to the abundance of Java APIs and Google libraries available.

Several approaches to detect sources and sinks have been proposed in the Android research community [51,62–65]. All these works rely on either program analysis or language specification to find candidate methods from a large collection of methods. These techniques present limitations when dealing with proprietary and obfuscated code, which raise the complexity of automated code analysis [66]. In Chapter 6, we propose an alternative approach to detect sources and sinks based on Android documentation and Natural Language Processing.

The following sections provide a comprehensive overview of the techniques used in this thesis. We first look into Program Analysis techniques to track sensitive data and the challenges in modelling Android apps. Then, we give a glimpse of contextual distributed representation in NLP and how we use this technique to model Android documentation.

2.3 Analysis Techniques

Program Analysis encompasses the techniques to reason about a program's properties automatically. For instance, we can use these techniques to detect bugs, determine the state of the input, or verify if a program complies with security/privacy requirements [15,31,41,67,68]. Program analysis is divided into two categories: static and dynamic analysis. While static analysis allows us to reason about program behaviour without running them, dynamic program analysis studies actual program executions.

There are several trade-offs in static and dynamic program analysis. First, static analysers are usually capable of examining the entire program code. Dynamic analysis often fails to reach full coverage due to incomplete test cases, resources and time limitations [31]. However, dynamic analysis shows evidence of actual executions, while static analysis only approximates real program behaviour. This means that a static analyser can only indicate potential data leaks which have to be confirmed either by manual inspection or dynamic analysis. One limitation of dynamic analysis is that malicious applications can detect testing environments and evade detection by hiding their behaviour [67,69]. The challenge in both cases is ensuring a sound analysis that is useful in practice.

As apps have become widespread in everyday tasks, vulnerable and malicious apps pose a considerable threat to final users and companies. Several static analysers have been developed by the research community [32–37] and the industry [70,71] to uncover vulnerabilities, find bugs, and detect potential data leaks. To name some examples, CHEX [72] is a tool to detect component hijacking vulnerabilities by analysing flows from external interfaces. Androbugs [73] uses a myriad of static analyses such as string analysis and program slicing to search vulnerabilities,

bad development practices, and dangerous shell commands. Overall, static analysis frameworks allow security practitioners to inspect thousands of apps without having to run them.

In this thesis, we rely mainly on static analysis to study information flows, with the exception of chapter 5, where we use both approaches as complementary techniques for the Android TV evaluation. Nevertheless, we briefly overview some dynamic analysis works before describing the principles of static analysis, with a focus on information flows. Enck *et at.* proposed a dynamic taint tracking approach for tracking sensitive data [46]. Their approach instruments the Dalvik environment and it is transparent for third-party apps. Sun *et al.* developed a similar framework for the new runtime environment (ART) [45]. Zhou *et al.* use a combination of firmware analysis, network traffic interception, and black- box testing to understand the interaction across IoT devices and mobile apps [15]. Other works stimulate apps to trigger and detect specific behaviours [49], use security testing to find vulnerabilities and data leaks [74], or analyse evasion techniques for dynamic analysis frameworks based on sensor data and virtual machine properties [67].

Overall, the research community uses static and dynamic analyses to study Android apps. We chose to evaluate different platforms using mainly static analysis to avoid the complex testing environments (e.g., devices) required to study multi-platform apps.

2.3.1 Static Analysis

A static analysis tool parses a program code and transverse program paths to check some property [75]. The input can take the form of source code, binary/byte code, or an intermediate representation. We can use static analysis to determine the state of a program, such as variables, pointers, definitions, and more. A data flow analysis examines a program to provide global information about how specific data elements are manipulated, e.g., compute the set of possible values at every program point [76]. A problem that is solved from the beginning of the program to the end uses *forward analysis*. The opposite approach, *backward analysis*, solves the problem from the last statement to the beginning. We use both techniques, e.g., forward analysis to detect potential data leaks and backward analysis to find the set of statements that affect a program point. A static analysis tool encounters the trade-off between being conservative and aggressive while never misrepresenting the program being analysed [76]. That is to say; it is desirable to detect all sensitive data flows but also keep the false positives as low as possible.

Analysis Sensitivity

Android apps are developed using Object-Oriented languages, which allow complex interactions between objects and method calls. The sensitivity of the analysis determines the precision of how these interactions are modelled [75]. The most common sensitivity levels for object-oriented programs are flow, context, object and field sensitivity [31]. A flow-sensitive analysis keeps track of the order of the statements, while a flow-insensitive approach does not consider the order. Context-sensitivity indicates how call sites are modelled. This can refer to the method call site or the object allocation site. A context-sensitive analysis is aware of a method

calling context by computing separate information for different calls to the same method. In contrast, a context-insensitive approach adds all possible call sites giving conservative results. Similarly, an object-sensitive approach uses object instances to differentiate contexts, while an object-insensitive approach does not distinguish between different instances of the same object. Last, field-sensitive analyses model each field of each object as a different abstraction, whereas a field-insensitive approach only models fields based on each object type.

2.3.2 Taint Analysis

Taint analysis is a data flow analysis that aims to find a connection between sources and sinks. Section 2.2.1 defines sources as predefined methods that read non-constant values from shared resources, and sinks as methods that write non-constant values to shared resources. As we are interested in sensitive data flows, we further limit our analyses to sources that read private data of interest, e.g., unique identifiers, location. Thus, we use taint analysis to determine confidentiality properties in Android apps.

The code snippet in Listing 2.2 shows a similar version of the code leaking the location in Listing 2.1. In this case, the location data is read in the `OnCreate` method, and propagated to the `sendData` method in line 4, where it is leaked. We use this example to get the general idea of taint analysis or taint tracking, and then we go through the details of the specific framework we use.

```
1 public void onCreate(){
2     Data text = new Data()
3     text.sensitive = Location.getLastKnownLocation()  //source
4     sendData(text)
5     }
6
7 public void sendData(Input data){
8     HTTP.post("www.analytics.com",data)  //sink
9 }
```

Listing 2.2: Taint Analysis example

We first need to represent the input program using control flow abstractions. A `Control flow graph` represents program statements in basic blocks and the flow among them. A `Call graph` is a directed graph where the nodes represent methods, and the edges represent call chains. The `Inter-procedural Control Flow Graph` (ICFG) combines the information from the Call graph with the Control flow graph. Auxiliary abstraction might be constructed on top of these structures.

Taint analysis uses a list of sources to taint variables that read sensitive data, e.g., the field `text.sensitive` in line 3. After that, the analysis propagates taints using a system of equations and control flow abstractions until a sink is reached, the taint is removed, or the propagation cannot continue. In our example, the tracking propagates the tainted information flow through the control flow abstraction from the method `onCreate` to `sendData`, and reaches the sink

method `post` in line 8. The data structure and algorithms to track tainted data vary. The following section provides the details of one popular solution, the IFDS framework [77].

Open-source frameworks.

A survey by Li *et al.* identified 38 frameworks for data flow analysis [75]. From this list, two independent studies [78, 79] conclude that three frameworks stand out for their soundness and performance: FlowDroid [32], Amandroid [33], and DroidSafe [34]. While no single tool provides the best performance in all cases, FlowDroid gives a good accuracy/running-time trade-off while being the only tool actively being updated. Therefore, we chose Flowdroid as a base tool to run taint tracking for Android apps. Another factor influencing this decision is that FlowDroid is built on top of Soot [80], a static analysis framework that we use for instrumentation, deobfuscation, and string analysis. Therefore, it is easier to integrate our frameworks with FlowDroid than other tools.

Flowdroid is a flow, context, field, and object-sensitive taint tracking framework. FlowDroid allows custom lists of sources and sinks and allows to specify shortcuts that model specific Android APIs (taint-wrappers). The IccTA extension enables FlowDroid to propagate communications across components [35]. FlowDroid inherits all Soot limitations. For instance, Soot might produce incomplete call graphs, particularly for new Java features [81]. However, the fact that the FlowDroid and Soot communities are active and related development groups ensures that problems can be treated in the future.

FlowDroid uses the Inter-procedural Distributive Subset Problem (IFDS) framework [77] to solve the taint tracking problem. The IFDS is a general framework that reduces an inter-procedural data flow problem, taint tracking in our case, to a graph reachability problem. We use the following code snippets to describe the IFDS procedure.

```
public OnCreate(){
    x = 1
    y = read(Location)
    y = sendData(x,y)
    leak(y)
}
```

Listing 2.3: Simplified code 1

```
public sendData(a,b){
    leak(b)
    b = a
    return b
}
```

Listing 2.4: Simplified code 2

Consider the control flow graph of the methods (Figure 2.7) and then the ICFG of both methods. The IFDS further transforms the ICFG into the *Supergraph* of the program (Figure 2.8). The Supergraph differs with a traditional ICFG in the following ways: 1) A method call is represented by two nodes (in grey), `call` (c*) and `return-site` (r-s*). 2) Each method call has three edges: A `call-to-return-site` edge (c-r-s*) to connect the call and return-site nodes. A `Call-to-start` edge (c-t-s*) to connect the call node with the called method. A `Exit-to-return-site` edge (e-t-r*) to connect the called method with the return-site node. These extra nodes and edges propagate local information to the global state of the program.

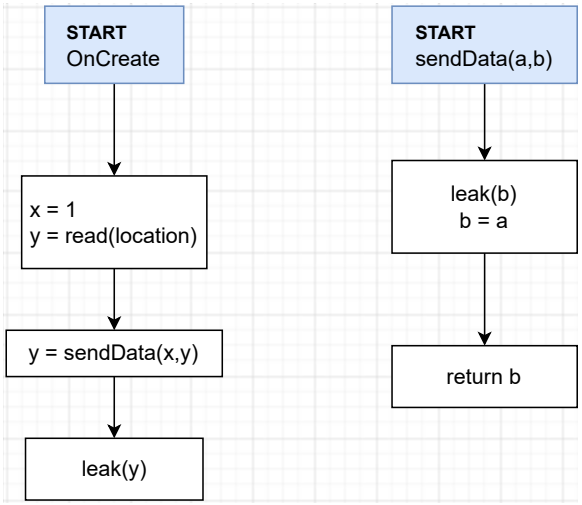


Figure 2.7: Control Flow graph of onCreate (left) and sendData (right)

The IFDS again transforms the Supergraph into a `Exploded Supergraph` using flow functions. These functions indicate the rules to propagate incoming facts based on the current statement. Figure 2.9 illustrates three traditional flow functions. The function `id` retains all incoming flow facts. For instance, if a variable is tainted, this function propagates the taint (for **a** and **b**). The node 0 represents a fact that holds unconditionally, usually the first statement of the program. The function `gen/kill` generates new flow facts for **a**, and discards incoming facts for **b** (e.g., a taint is added for a and removed for b). The last function is a combination of the two previous functions. To generate the Expanded supergraph, each abstraction from the Supergraph is turned into a node of the exploded Supergraph. Then, the data flow functions are encoded as edges between the nodes representing facts at different program points.

Figure 2.10 shows the generated Exploded Supergraph. Let us use the first and second instructions to understand how the Exploded supergraph is constructed. The `x = 1` assignment kills any previous taint, represented by the missing arrow in x. In this domain, constant values are not relevant. The `y = read(location)` instruction kills any incoming fact and taints the `y` variable by adding an edge from the special variable ˆ that represents the unconditional fact or node 0. The procedure continues until every instruction is computed. Note that the local and global facts are propagated back and forth through the edges connecting the special nodes.

This representation allows us to verify if a fact holds at different program points. Thus, the problem of taint tracking is reduced to computing reachability relationships between nodes of the Exploded supergraph. For instance, we can follow the tainted variable `y` and reach the instruction that calls `leak(b)` in the method `sendData`, indicating a potential data leak. However, there is no tainted value in the second call to the method `leak(y)` in the method `onCreate`.

FlowDroid uses an optimisation proposed by Naeem *et al.* where the exploded Supergraph is created on demand [82], instead of creating the entire structure before calculating the connection between sources and sinks. As consequence, the unconditional fact represents a source call, instead of the standard first program statement.

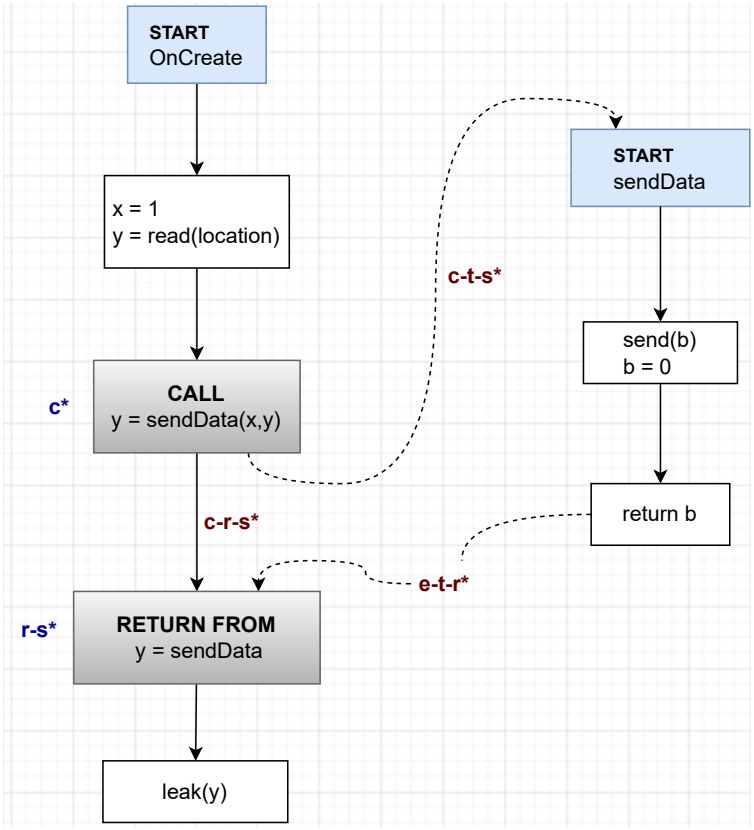


Figure 2.8: Supergraph constructed by the IFDS framework

Encoding a problem in the IFDS framework may result in precision loss due to aliasing. For instance, when a tainted value is assigned to arrays or object fields, all the objects that point to the same address in the heap must also be tainted to ensure completeness. Flowdroid runs a backward alias analysis when tainted values are assigned to the heap to address this limitation. Last, Flowdroid does not always terminate due to the undecidability of static analysis [83], even when simplifications and assumptions are in place.

We conclude this section by describing how we use taint tracking. In Chapter 4, we study sensitive information flows between mobile and wearable apps to detect data leaks. Our framework models wearable APIs and includes string analysis, backward slicing, bytecode instrumentation, and de-obfuscation to handle different challenges. We embed FlowDroid in our framework

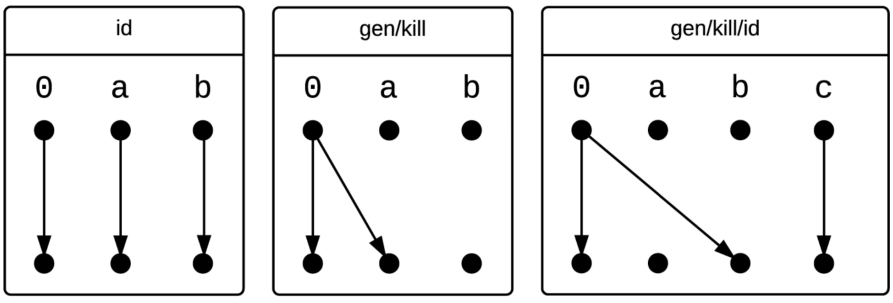


Figure 2.9: Flow functions (reproduced from [77])

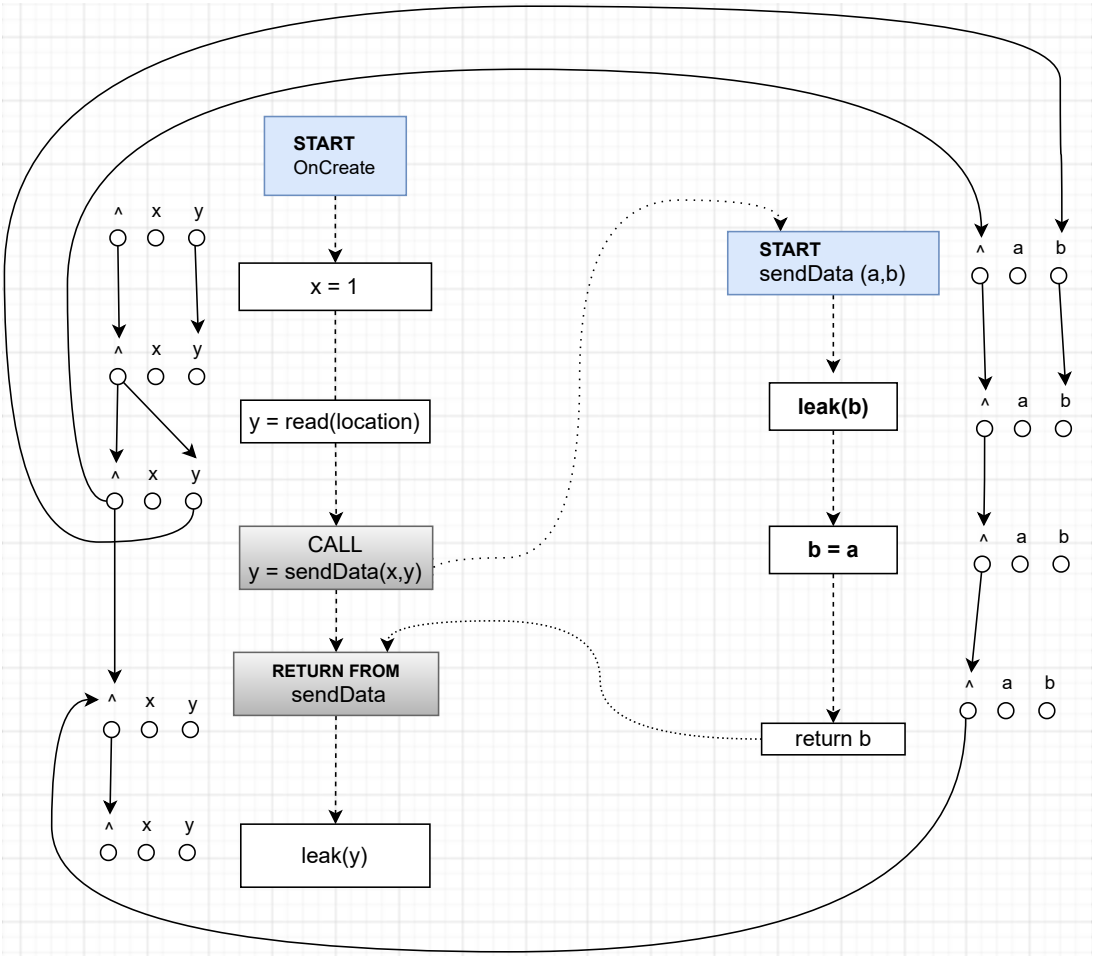


Figure 2.10: Exploded Supergraph

`WearFlow` and build other analyses on top of Soot. In Chapter 5, we study sensitive data leaks using several static analyses, including taint tracking with FlowDroid. We complement this with dynamic analysis and traffic analysis to monitor TV app behaviour in Android TV.

2.3.3 Challenges of Modelling Android Apps

Study information flows in Android apps present some challenges due to their architecture (Section 2.1). We briefly discuss some these challenges.

Code translation and Intermediate Representation (IR). Android applications are compiled into Java bytecode and then converted to Dalvik bytecode. Apps source code is usually unavailable because our thesis targets third-party apps. Even if source code is available, using an IR for static analysis has benefits: 1) The instruction sets are simpler and optimised for analyses. 2) IR can abstract from concrete input languages. In this thesis, we use Soot [80] and Androguard [84] to decompile apps and transform them into a convenient IR.

Soot is a static analysis framework that provides various functionalities, including the translation of Java and Dalvik bytecode into IRs, and elemental static analyses (e.g., aliasing and constant value propagation) that can be used to solve more complex problems. Soot uses Jimple as its

default IR. Jimple is a register-based IR with only 15 instructions, highly optimised for code analysis. For instance, it provides local variables, no nested instructions, and special variables for the reference `this` and parameters. In contrast, Dalvik bytecode offers a richer instruction set, with 237 opcodes, albeit optimised for running time. We use Jimple for APK instrumentation, de-obfuscation and taint tracking in Chapter 4.

Androguard is a tool for reverse engineering Android apps that comes with its own IR. Androguard lacks Soot code transformation capabilities, but it is more convenient for quick analyses, such as classes, cross-references and Android Manifest. We use Androguard in Chapter 5 for manual and automatic static analysis, and in Chapter 3 to study differences between apps across different platforms.

Entry-points. Android apps do not have a main method, so there is no unique starting point. A control flow abstraction could use any component in the Manifest as entry point. FlowDroid and other frameworks [32,35] solve this issue by creating a main dummy method and then connecting declared component to it. Other frameworks use domain knowledge and Android documentation to model entry points [34, 85, 86], and others iterate over frameworks methods to generate the call graph [72]. Similarly, callbacks and component lifecycle methods add complexity to the construction of control flow abstraction. These are usually modelled individually.

Libraries and APIs Detection. Third-party libraries (TPL) are common in Android apps as developers use them for different purposes, such as monetisation, tracking, social media, or simple utilities. Even though they facilitate the development process, these libraries have been shown to exacerbate security and privacy risks to their host apps [19,21,22]. Thus, studying TPL is fundamental to security and privacy in Android platforms. To give one example, Reardon et al. found that libraries such as Baidu and Salmonads used covert channels to collect sensitive data [47]. We are interested in detecting abusive behaviours but also want to attribute such behaviours to the app or library code. To do this, we first need to identify TPL and then associate package names to libraries.

Several open-source tools exist to identify TPL, which offer different capabilities [19–22]. However, no single tool provides the best results. We chose LibScout [19], and LibRadar [20] for this task. The former is a lightweight obfuscation-resilient library detection tool that generates library profiles from binaries. The latter generates library profiles by analysing code features from a large dataset of Android apps. LibScout reduces the false positive rate by profiling from binaries, while LibRadar covers a broader range of libraries at the cost of neglecting the ground truth. We also develop a custom approach to complement results from these off-the-shelf tools for our evaluation in Chapter 5. Our custom approach allows us to detect libraries for which we do not have a profile available.

Obfuscation. Obfuscation techniques are becoming increasingly popular across Android platforms [40,66]. Benign developers tend to obfuscate their APKs to protect their code and optimise resources. Malicious developers use obfuscation techniques to hinder scrutiny from automated tools and security practitioners. Hammad *et al.* study the effects of obfuscation on Android

apps and anti-malware products [42]. This work separates trivial from non-trivial obfuscation techniques. The main difference is that trivial techniques do not modify the app bytecode and focus on metadata and signatures (e.g., alignment or Manifest transformation). Non-trivial techniques modify app bytecode and present a challenge to static analysers because it could modify the signature of methods and hinder the detection of libraries. We list the most popular non-trivial obfuscation techniques according to the aforementioned work.

- *Identifiers renaming.* Aims to remove semantic information by replacing identifiers and class names with meaningless text. Code written following good development practices uses meaningful names to improve readability. A side effect is that these good practices ease the job of reverse engineers and automated tools. This obfuscation technique strips the semantics from the source or byte code.

- *String encryption.* Obfuscating strings literals seeks to remove their semantic information from the code. Instead of using hard-coded strings, developers use cryptographic functions or less sophisticated methods to encrypt string literals.

- *Reflection* is a Java feature that invokes classes and objects dynamically. While this is a feature in the Java language and not an obfuscation technique, it allows a program to manipulate internal properties of the app. Thus, reflection can be used to obfuscate the invocation of sensitive methods, e.g., sources and sinks.

- *Control flow manipulation.* This technique seeks to modify the control flow graph of the app by adding iterative or logical instruction. Additionally, it is possible to add method calls that modify the original call graph.

- *Junk code.* This technique inserts instructions that do not affect the app execution but modify the code signature and can fool static analysers.

Developers might combine these strategies to produce stronger obfuscated APKs. Dong *et al.* present a large-scale investigation of common obfuscation techniques and their prevalence in Android apps [66]. They found that string encryption and complex renaming policies are more prevalent in malware. Reflection cases are mostly used to invoke hidden functions or backward compatibility. WearFlow (Chapter 4) has a de-obfuscation module that uses type signatures and abstract types to generate obfuscation-resilient signatures to detect security-sensitive methods. We provide all the details in the corresponding chapter.

Incomplete Environments for Information Flows. Application developers use platform features that can affect the accuracy of static analysers. For instance, obfuscation and Java Reflection can hinder the detection of API methods. DroidRA [40] solves the Reflection problem by reducing it to a composite constant propagation problem and instrumenting the APKs with the reflective targets. Another problem is dynamic code updates. Poeplau *et al.* proposed a static analysis approach based on program slicing [87] to detect dynamic code updates [88]. StaDynA [89] is a tool that uses static and dynamic analysis to resolve reflective calls and dynamic code loading for incomplete environments. The tool complements the app's call graph

with the dynamic analysis results. Similarly, static analysis for multi-language applications is more complex than single-language programs. Wei *et al.* proposed an efficient inter-language model to propagate data flows across Java and native code [41]. JuCify [38] is a tool that first generates the call graph by analysing Java bytecode and adds additional edges by analysing native code. BabelView [90] and BridgeScope [91] are frameworks for propagating data flows from Java to JavaScript code in WebViews. Even though this thesis does not focus on solving these issues, our work can be used along these frameworks to provide a better understanding of app behaviour.

Network Traffic Analysis. The preceding discussions are related to static analysis. While most of our experiments analyse apps statically, we also run dynamic analysis experiments to monitor TV apps' network traffic in Chapter 5. Identifying sensitive information in Android TV traffic presents some challenges. First, apps usually send encrypted traffic to the servers. Second, some apps employ advanced techniques to protect the traffic, such as certificate pinning. Last, sensitive data collection or abusive behaviours might only appear after a specific event. For instance, trackers tend to increase their activity after the users have authenticated [29]. We address all these problems in details in Chapter 5.

2.4 Software Documentation

Modern software projects come with documentation that provides information about the correct use of components, e.g., functions or APIs. Software documentation can take many forms. For instance, API documentation, manual pages, and code comments serve the abovementioned purpose. In this thesis, we focus our attention on API documentation, which helps developers associate API methods' semantics with the corresponding code. API documentation is expressed in a natural language, while Android code is expressed in a programming language. The cooperation between these two channels forms a dual channel [92] view that provides an opportunity to enhance the understanding of software components based on documentation.

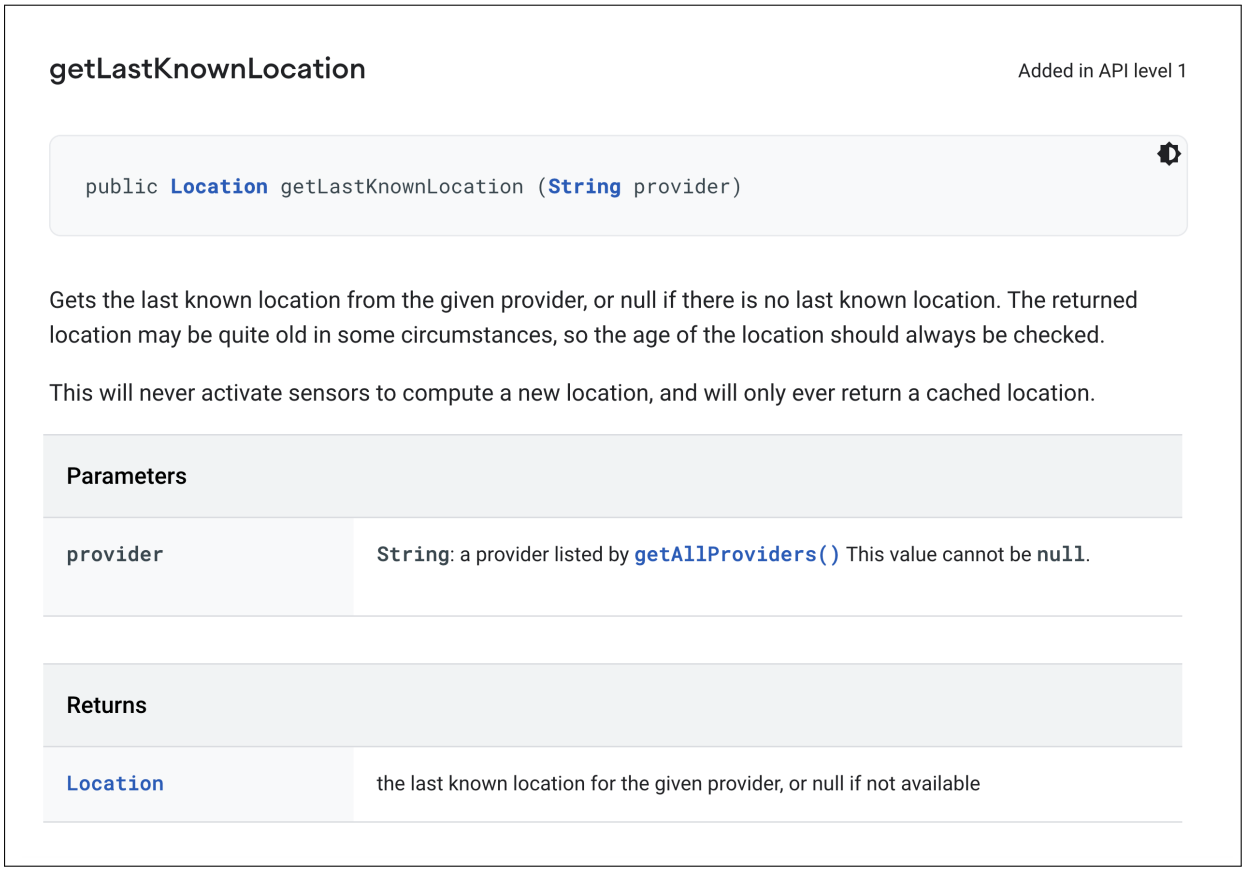
APIs are a powerful mechanism that enables complex functionalities and the reuse of components. However, use them properly is far from trivial. In this context, documentation has been described as instrumental to the success of software that relies on APIs [53]. For instance, representative names, perceptible relations between API types, and accurate descriptions are fundamental for good API usability [93]. One such example is the Android platform, which is intended to be used by millions of third-party developers and provide high-quality documentation.

2.4.1 Documentation in Android

App developers rely on Android, Google libraries, and third-party libraries documentation to support their development effort. We focus on Android and Google libraries documentation, even though the same principles apply to third-party libraries documentation (especially popular libraries with a large user base). The Android architecture is designed for reusing components accessed through their APIs (Section 2.2.1). Thus, its source code is accompanied by rich and

detailed documentation. This makes Android API documentation a good candidate for using a data-driven approach to infer semantic properties of the underlying implementation.

Android and Google libraries documentation is available in the source code as Javadoc or on the developer's website. A method documentation describes the purpose, the inputs and outputs, and sometimes other considerations such as permissions or recommendations. Figure 2.11 shows the documentation of the `getLastKnownLocation` method used in Listing 2.2. We show the online documentation for reference, and we revisit this example in Chapter 6. This illustrative example clearly shows that Android documentation is a great source for extracting syntactic and semantic information about API methods.



getLastKnownLocation Added in API level 1

```
public Location getLastKnownLocation (String provider)
```

Gets the last known location from the given provider, or null if there is no last known location. The returned location may be quite old in some circumstances, so the age of the location should always be checked.

This will never activate sensors to compute a new location, and will only ever return a cached location.

Parameters	
`provider`	`String`: a provider listed by `getAllProviders()` This value cannot be `null`.

Returns	
`Location`	the last known location for the given provider, or null if not available

Figure 2.11: getLastKnowLocation method description

We argue that API documentation provides a rich source of information to understand the purpose of methods and their relevance to security analysis. We take advantage of the semantic information provided by Android documentation and propose an NLP framework to classify Android methods using their documentation in Chapter 6. This section describes text representation techniques in NLP, and we leave the details of our framework to the corresponding chapter.

2.4.2 Text Representation in NLP

Text classification is a primary NLP task used for information retrieval, ranking, inferences, sentiment analysis, and others [94, 95]. Text classification aims to assign a label or category to

an input text or document. The input text, expressed in natural language, has to be encoded to feed a Machine Learning model. In other words, we first need to convert words into a numeric representation. Sparse and dense vectors are two options for achieving this encoding. Next, we describe the mainstream encoding techniques and their limitations using the two sentences below.

$$s1 = \text{This is sentence one}$$
$$s2 = \text{This is sentence one plus this new word}$$

The *one-hot encoding* technique associates each token to a vector element, where categorical features are represented in the *index*. The index consists of the vocabulary of the input text and a positional integer. The index for our two sentences is shown below.

$$index = \{sentence : 0, plus : 1, this : 2, one : 3, is : 4, new : 5, word : 6\}$$

A sentence is represented by a sparse matrix $(N * M)$ where N is the number of tokens and M is the index's size. Each token from a sentence is represented by a vector where a token occurrence is marked by a 1 in its corresponding index position.

The one-hot encoding representation of s1 and s2 is shown below.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

s1: This is sentence one

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

s2: This is sentence one plus this new word

There are two problems with this representation. First, the size of the matrix increases as the training corpus gets bigger. Second, the sparse matrix size is proportional to the length of the input sentence. These problems have to be addressed because most classification models expect inputs of the same size, either by encoding characters instead of words or by making the size of the representation fixed, albeit losing information [96].

There are some alternatives to one-hot encoding which address these problems. For instance, a **count-vectoriser** represents sentences based on word frequencies instead of occurrence (each word is represented by its frequency). With this, a sentence is mapped into a single vector, e.g., $s2 = [1, 1, 2, 1, 1, 1, 1]$. This technique optimises the vector size at the cost of losing ordering information. Other alternatives, such as **Tf-ids** and **n-gram** vectorisers, modify the frequency formula or the number of consecutive words that form a token, e.g., 2-gram uses 2 words to form a token. Overall, traditional NLP encoding techniques lack the capability of modelling the order of words and their semantic efficiently [97]. While these techniques might be useful in some scenarios, they are ineffective for tasks that require understanding the context of sentences

[96, 97]. Moreover, there is no similarity measure to compare words or sentence semantics directly [98]. For instance, the words "car" and "automobile" are statistically orthogonal.

Recent advances in NLP enable the training of Deep Learning models using large corpus with the aim of capturing the semantics of words. Hinton *et al.* were the first to propose the word embedding technique that generates distributed representation of words by mapping text to high dimensional vectors [99]. This representation aims to capture the word semantics in vectors of real numbers. That is, the meaning is "distributed" across multiple components instead of local representation where each element represents exactly one component [98].

Word2Vec, proposed by *Mikolov et al.* in 2013, was a ground-breaking model that produced state-of-the-art performance at low computational cost [97, 100]. Word2Vec algorithms train a 2-layer neural network on large corpus of unlabelled data (e.g. a Wikipedia dump) using a Masked Language Model task. The objective for this task is to predict a word giving its context. Figure 2.12 depicts the two Word2Vec training modes: In the **CBOW** model, the objective is to predict a target word using context words within a window (w). In the **Skip-gram** model, the objective is to produce context words given the target word.

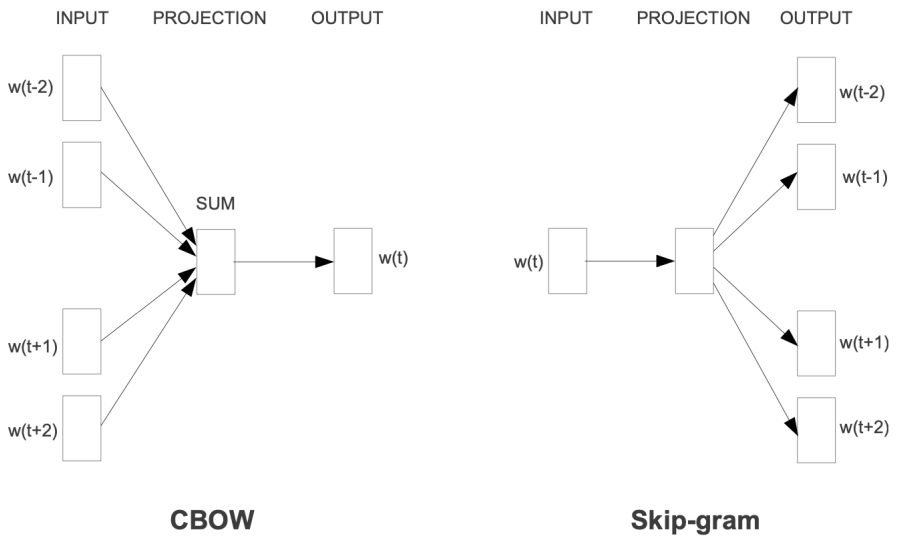


Figure 2.12: Word2Vec CBOW (Continuous bag-of-word) and Skip-gram models (adapted from [97])

Word2Vec fails to produce contextualised embedding for the same word in different contexts. For instance, the word bank can have different meanings depending on the context (e.g., financial institution or the land alongside a river) To address this problem, Devlin et al. proposed a deep bidirectional network BERT (Bidirectional Encoder Representations from Transformers) [101] that is pre-trained using unlabelled data and then fine-tuned with multiple supervised tasks. For the pre-training phase, BERT uses the BookCorpus (800M words) and English Wikipedia (2500M words) as the corpus, and two unsupervised tasks: masked language (similar to Word2Vec) and next sentence prediction. The BERT transfer learning approach enables the reuse of expensive pre-trained networks with a much cheaper fine-tuning classification tasks.

BERT improves previous general language representation that use unidirectional language models by using left and right context during training across all layers.

Still, BERT suffers from computational overheads when used to compare embeddings and clustering. In this thesis, we use primarily Sentence-BERT [102], a modification of the original BERT network that uses siamese and triplet network structure to produce sentence embeddings that enable more efficient search and clustering operations. Figure 2.13 shows the Sentence-BERT tuning architecture. In this figure, BERT is a multi-head attention network [103] (as described above) that connects to a pooling layer to derive semantically meaningful fixed-size vector embeddings (768 dimension by default). The network is then fine-tuned with several tasks depending on the available data.

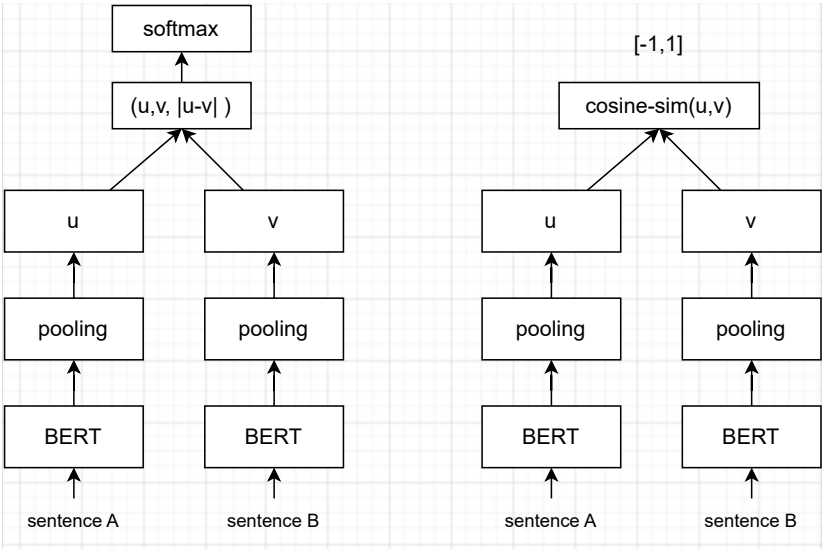


Figure 2.13: Fine tuning Sentence-BERT Classification (left) and Inference (right) architectures

The best network architecture depends on the available data. In our evaluation (Chapter 6), we use the Inference architecture to fine tune the network based on pairs of Android methods and a corresponding similarity score using the Semantic Textual Similarity task. Reimers *et al.* explored different metrics such as cosine similarity, euclidean distance, and dot product to compare embeddings and conclude that all lead to similar results [102]. In our evaluation, we use the cosine similarity metric defined by:

$$Cosine(x, y) = \frac{x \cdot y}{|x||y|}$$

The idea of using NLP techniques to study apps has been used before to analyse app metadata [104, 105], permissions [106, 107], reviews, and privacy policies [108–111]. Previous works highlighted the importance of software documentation to extract semantic information about programs [112, 113]. However, little attention has been given to API documentation to detect sensitive methods for security analysis, and current approaches rely on program analysis for this task [51, 62–65].

Sentence-BERT architecture allows us to make inferences from Android API documentation efficiently. In Chapter 6, we build a classifier that allow us to detect security-sensitive methods and run similarity queries in a corpus of Android API documentation. Our framework, `DocFlow`, enables the generation of taint tracking specifications based on the semantics of Android documentation. This semantics is modelled by Sentence-BERT and complemented by other techniques that we explain in the corresponding chapter. To evaluate the classifiers, we use the metrics defined by the following equations:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \qquad Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN} \qquad F1 = \frac{2 * TP}{2 * TP + FP + FN}$$

Consider a binary classifier that predicts if a method is a source or not-source (the example can be extrapolated to a multiclass problem). Assume that the source class is the positive and not-source the negative class. The correct predictions are the true positives (TP) and true negatives (TN), while the incorrect predictions are the false positives (FP) and false negatives (FN). Thus, the *accuracy* indicates the fraction of correct predictions. However, this metric can be misleading, particularly for an imbalanced dataset [114]. The *precision* indicates the proportion of correct positive predictions, and the *recall* indicates the proportion of positive samples correctly detected. The *f1-score* is the harmonic mean of the precision and recall and can balance the trade-off between both.

2.5 Chapter Summary

Android apps provide an interface with the digital world. These apps are mainly written in Java/Kotlin, compiled to Dalvik bytecode and packed into APK files. Android apps rely on a tightly coupled architecture with the Java framework, which offers access to lower layers. We have described how apps use the Java Framework to access shared resources and the Android security model. Sensitive methods (sources and sinks) allow apps to access and expose sensitive data to other parties.

In this chapter, we have presented the techniques we use to study multi-platform apps in Android. Our approach is based on a dual-channel perspective [92], where Natural Language Processing and Program Analysis cooperate for software security analysis. We focus on information flow analysis techniques, text representation and language models in NLP. We have described the limitations and challenges of modelling Android apps with these techniques. Now we continue by analysing structural differences and threats across Android platforms. The following chapters present the contributions of this thesis and answer the research questions.

3 Apps Ecosystem Characterisation

In this chapter we study the commonalities and unique characteristic of Android apps across platforms. This characterisation allows us to understand the different threats specific to each platform and develop solutions to address the uniqueness of each Android platform.

Smart devices evolved from small devices with reduced computation and connection capabilities to more advanced devices with enhanced capabilities, such as smartwatches, smart TVs, and smart cars. Figure 3.1 illustrates a typical daily scenario where users interact with multiple smart devices. To offer a perspective, the number of connected IoT devices is expected to reach 18 billion in 2022 [115]. Similarly, the Smart Home market is predicted to expand at a compound annual growth rate of 27% from 2022 to 2030 [116]. Undoubtedly, interconnected ecosystems are ubiquitous in everyday life.

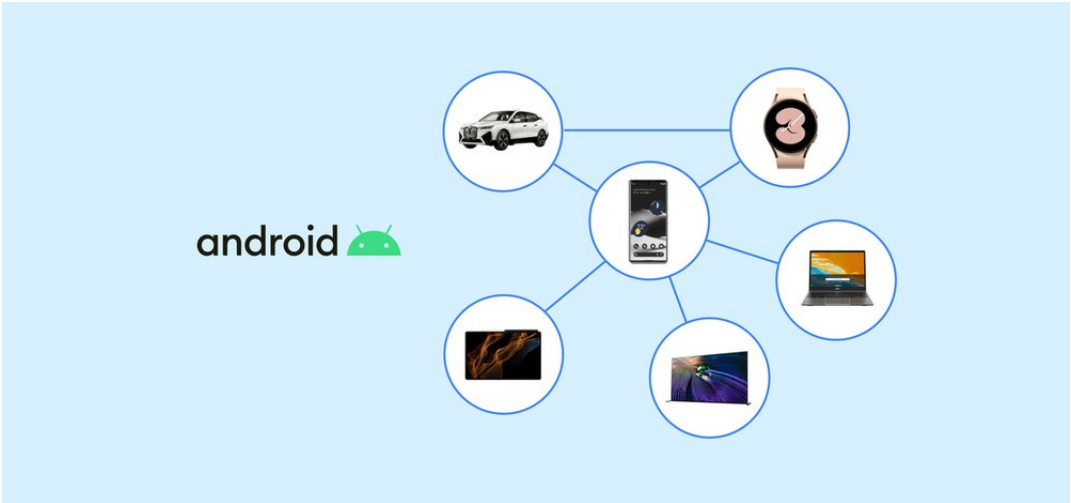


Figure 3.1: Android platforms (Adapted from [117])

To better understand this scenario, Zhou *et al.* define three main interacting entities: cloud services, smart devices, and applications [15]. Each of these entities plays a role in the security and privacy of the ecosystem. This thesis focuses on applications and their interaction across Android platforms. Note that our work targets applications that run on high-end Android devices (e.g., smartphones and smart TVs) rather than small IoT gadgets. Thus, applications for IoT or medical devices are outside the scope of this thesis. A separate line of research deals with commodity IoT devices and their security and privacy implications [16, 118–120].

Google aims to expand its vision of cooperating multi-device ecosystem where each device offers an enhanced experience [117]. While smartphones are the most popular device, other devices use a customised version of Android. For each platform, Google considers the execution context,

the range of smart devices, and the user experience. We list some of these devices below.

- Tablet devices: Tablets are provisioned with the same Android version as smartphones.
- Smartwatches: Wear OS is a stripped version of Android optimised to run wearable apps on Android smartwatches and designed for a wrist experience.
- Smart TVs: Android TV is a customised version of Android to support specialised TV hardware, and it is designed to provide a TV experience.
- Smart Cars: Android Auto is a version that allows users to connect their phone to a vehicle to display a customised app version on the vehicle console.

We chose to study Wear OS (Chapter 4) and Android TV (Chapter 5) apps as an initial approximation to analyse apps on arbitrary platforms. The contributions from these chapters allow us to understand the limitations of current techniques to detect sensitive data flows in arbitrary platforms. It is worth noting that these two platforms are more mature than others in terms of years of development, adoption, and the number of apps. We use the experience gained by analysing wearable and TV apps to propose a framework that can generate taint analysis specifications for arbitrary platforms in Chapter 6.

The remaining sections describe the following topics: 1) Google Play Services, 2) Wear OS and wearable apps, and 3) Android TV and TV apps. Although we leave Android Auto outside the scope of this thesis, we discuss potential future work in Section 7.2.

3.1 Google Play Services

While Android is an open-source OS, most "stock" Android devices run proprietary software from manufacturers (OEMs) and third parties [121]. To access the Google Play Store, Google requires device manufacturers to include other core modules such as Google Mobile Services (GMS). These services include Google apps (Maps, Youtube, etc.) and background services, also known as Google Play Services.

To understand this architecture, we need to go back to the initial years of Android. The Android ecosystem suffered a fragmentation problem as OEMs were unable to keep up with Google updates [122]. In response to the security issues underlying the fragmentation problem, Google moved the most critical components of Android to the Google Play Services bundle. This library receives automated updates from the Play Store without involving OEMs or users. Thus, Google Play Services is a fundamental part of all Android platforms, albeit largely ignored by the research community.

Google Play Services has two core components: 1) a proprietary app that embeds the logic of the different services offered by Google, and 2) a client library that provides an interface to those services. Developers must include the client library in their apps when accessing Google-dependent services, including those regarding Wear OS and some Android TV components. Figure 3.2

shows how the Google Play Services app interacts with the client library using standard inter-process communication (IPC) channels.

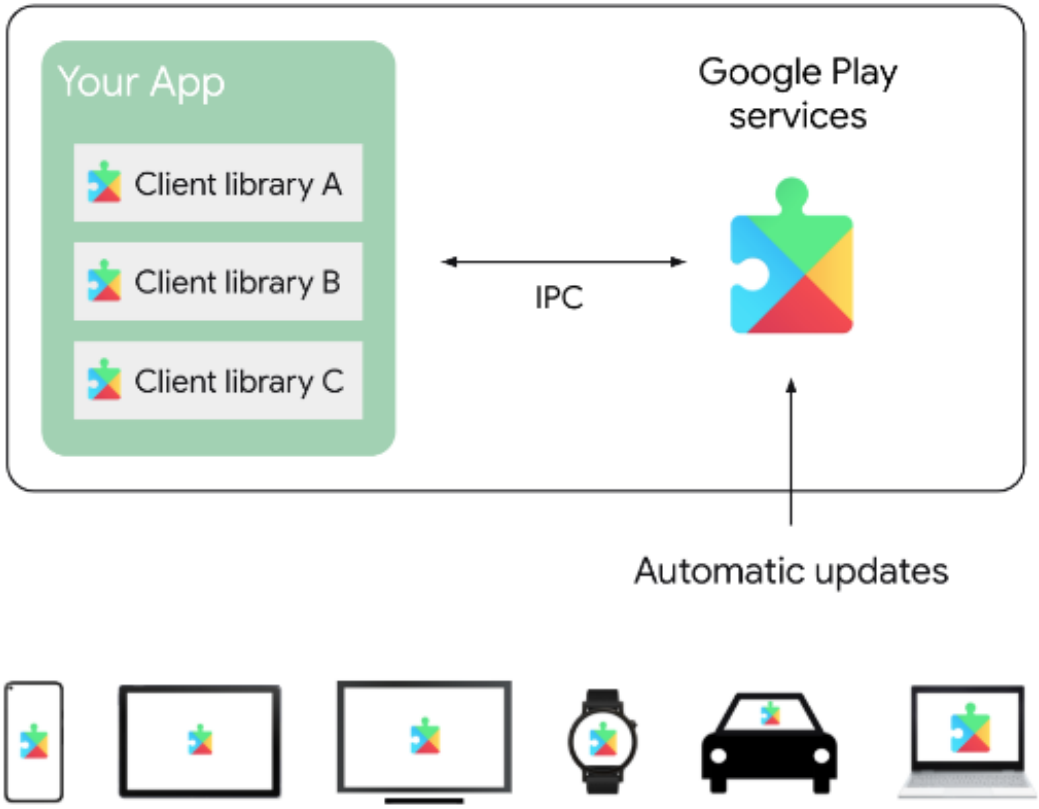


Figure 3.2: Google Play Services (GPS) architecture (extracted from [123])

Google provides more than 30 packages that allow developers to interface with all the Google Play Services as of February 2023. Table 3.1 shows a subset of these libraries and the supported devices for each library. The full list is available in the official documentation [123].

A Google Play Services API call is not different from other Android API from the developer's perspective. However, many Google Play Service APIs implement complex functionalities in the native app. Consider the Listing 3.1, and the call to `getLastLocation()` in line 6. The code immediately adds a callback to handle the result of this method instead of assigning the result to a variable. The semantics of this operation is lost for a static analyser that inspects this code. Therefore, a taint tracking analysis needs a model of this API call to properly propagate taints that use this API and other Google Play Services APIs. In particular, the Mobile-Wear communication uses the Data Layer APIs from Google Play Services that must be modelled for a sound analysis. We describe this communication type in the following section.

Library Name	Supported Devices
Google Mobile Ads	Phone, Tablet
Android Advertising ID	Phone, Tablet, Android TV, Chrome OS
Google Sign-In for Android	Phone, Tablet, Android TV, Chrome Os
Google Awareness API	Phone, Tablet, Android TV, Auto, Chrome OS, Wear OS
Google Cast API	Phone, Tablet, Chrome OS
Google Fit API	Phone, Tablet, Chrome OS, Wear OS
Fused Location Provider	Phone, Tablet, Android TV, Auto, Chrome OS, Wear OS
Google Maps SDK	Phone, Tablet, Android TV, Auto, ChromeOS, Wear OS
Nearby Platform API	Phone, Tablet, Android TV, Auto
Google Pay API	Phone, Tablet, Auto, Android Go, ChromeOS, Wear OS
Wearable Data Layer API	Phone, Tablet, Wear OS

Table 3.1: Subset of Google Play Services libraries

```
// Code required for requesting location permissions omitted for brevity.
FusedLocationProviderClient client =
        LocationServices.getFusedLocationProviderClient(this);

// Get the last known location. In some rare situations, this can be null.
client.getLastLocation().
        .addOnSuccessListener(this, location -> {
            if (location != null) {
                // Logic to handle location object.
            }
        });
}
```

Listing 3.1: Google Play Service location API example

3.2 Wearable Platform

Wear OS is a stripped version of Android optimised to run wearable apps on Android smartwatches. The capabilities of these smartwatches range depending on the hardware of the manufacturer. Apart from main components such as screen and CPU, these devices incorporate an array of sensors including accelerometers, heart-rate and pedometer among others. The Wear OS provides an abstraction for apps to access those sensors.

Wear apps are similar to mobile apps, but their design and functionality is tailored for critical tasks and a wrist experience. For instance, a fitness Wear OS app mostly focus on data gathering while leaving the data analysis and more complex features that require more screenspace to the mobile app. Wear OS offers complex interaction mechanism with the mobile counterpart. For instance, Figure 3.3 shows a standalone payment app and media player app that allows play media content across devices. Figure 3.4 shows a mobile clock app and its companion. These apps can synchronise data using wearable libraries from Google Play Services.

Wear OS adopts the same security model used to protect its mobile counterpart. Wearable

Figure 3.3: Standalone wearable apps

Figure 3.4: Mobile and wearable companion app

Figure 3.5: Example of wearable and mobile apps (Adapted from documentation [124])

applications are sandboxed and installed with minimum permissions by default. Dangerous permissions are granted at run-time. Permissions still need to be declared on the app Manifest. The only special consideration for Wear apps is the independent authorisation process as permissions are not inherited from the mobile app. The wearable app must request permission to access protected resources. These resources can be either in the smartwatch or in the smartphone (the smartwatch can also access resources in the smartphone and vice-versa, provided users grant the appropriate permissions).

3.2.1 Mobile vs Wearable Apps

We conducted a preliminary analysis to better understand the difference between mobile and wearable apps. We aim to compare the wearable version with the mobile counterpart in terms of size, permissions and third-party libraries. The permissions were extracted using Androguard, and we used LibRadar and LibScout to detect third-party libraries. We download a small number of wearable-enabled apps from the Play Store, selecting the most popular wearable apps.

Table 3.2 shows the package names and the difference between the versions. The results show that mobile apps are larger in size and functionalities and provide more functionalities based on permissions and libraries. 90% of Wear OS permissions are common to both platforms, as expected. However, the remaining Wear OS permissions enable functionalities inherent in smartwatches like wet-mode, watch-face control, and palm lock functionalities. In total, there are 15 unique Wear OS and 41 mobile permissions. We found 52 third-party packages that are unique to wearable apps and enable wear-specific features such as watch faces, complications, and custom authentication APIs for smartwatches. Notably, there are cases where app versions use different libraries for the same purpose. For instance, the wearable app `iheartradio` uses the analytics libraries Comscore and play-services-analytics, but the mobile version uses AppsFlyer and Firebase for the same purpose. This example shows that the same app can present different behaviour across platform versions.

Although this preliminary study is not exhaustive, it demonstrates differences between apps

	Mobile			Wear		
Apps	Size (MB)	Permission (AOSP-TP)	Libraries	Size (MB)	Permission (AOSP/TP)	Libraries
com.shazam.android	31	11-4	6	6.5	3-0	4
com.accuweather.android	130	10-6	22	7.5	3-1	6
com.amazon.mp3	76	23-30	49	67	6-0	14
com.callapp.contacts	47	46-14	55	4.2	4-1	6
com.clearchannel.iheartradio.controller	58	18-12	27	25	12-7	40
com.google.android.apps.maps	102	23-11	18	18	11-3	11
com.google.android.apps.walletnfcrel	17	18-5	23	5.3	10-6	8
com.microsoft.office.outlook	88	28-28	21	13	3-2	4
com.soundcloud.android	88	14-5	42	6.3	12-2	8
com.strava	72	19-15	15	20	11-7	8
com.todoist	43	9-7	19	7.5	10-7	4

Table 3.2: Popular Wear OS apps. Mobile and wearable comparison. The column Permission shows the difference between Android (AOSP) and (TP) third-party declared permissions.

across these two platforms. Wearable apps offer functionalities that are not available in mobile apps. This difference manifests in permissions and third-party libraries available only for Wear OS applications. These differences give rise to threats that should be considered properly. Moreover, the Mobile-Wear interactions result in new threats unknown in standalone settings that we describe below.

3.2.2 Communication in Wearable Apps

Wear devices are equipped with network connectivity like Bluetooth, NFC, WiFi, or even access to cellular networks. Most watches require a phone pairing process via Bluetooth or WiFi. The pairing process establishes a low-level channel that can be used by mobile apps to communicate with a companion app in the smartwatch. Note, however, that wearable apps can run standalone apps (i.e., no mobile app needed) from Wear OS 2.0. Figure 3.6 illustrates the interplay between a mobile phone, a smartwatch and the network. Note that the Google Play Services app uses the low-level channels (WiFi or Bluetooth), while the mobile and wearable apps call high-level wearable APIs that abstract the complexity of the communication.

The Google Plat Services package `com.google.android.gms.wearable` gathers all the interfaces exposed for wearable apps, including the APIs that enable the communication between mobile and wearable apps. This package is commonly referred as the `Data Layer` API. We next describe how Wear OS enable apps to communicate with each other, including how they communicate with the mobile companion app.

3.2.3 Data Layer

The `Data Layer` API provides IPC capabilities to apps. This API consists of a set of data objects, methods, and listeners that apps can rely on to send data using four types of abstraction:

1. `DataItem` is a key-value structure that provides automatic synchronisation between devices for payloads up to 100KB. The keys are string values, and the payload could be integers, strings or other 16 data types. The `DataClient` APIs offer support to send `DataItems` which

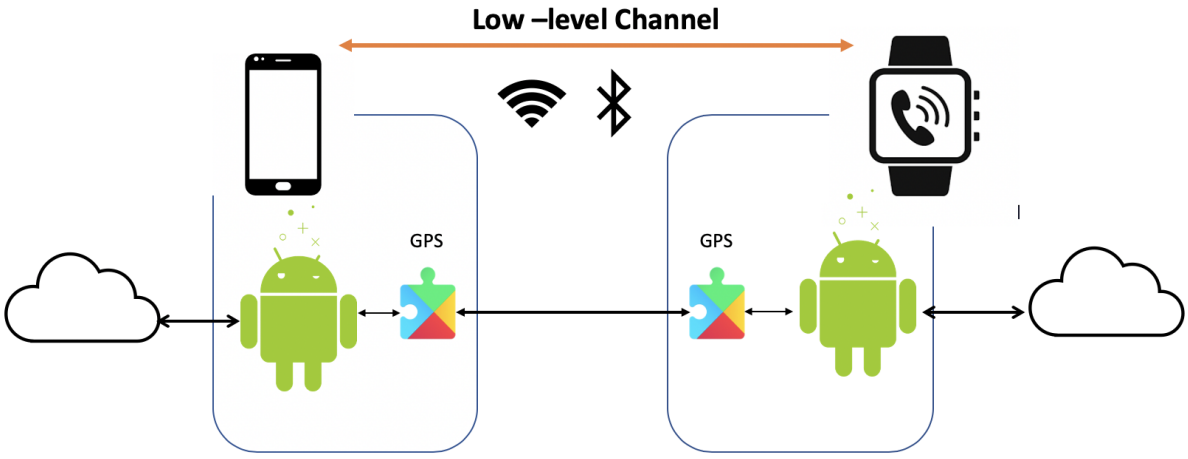


Figure 3.6: Communication between a mobile app, its companion, Google Play Services (GPS) and the network.

are uniquely identified by a path (string value) in the system.

2. `Assets` are objects that support large binaries like images or audio. Assets are encapsulated into `DataItems` before being sent. The `Data Layer` takes care of transferring the data, bandwidth administration, and caching the binaries.

3. `Message` are short bytes of text messages that can be used for controlling media players, starting intents on the wearable from the mobile, or request/response communication. The `MessageClient` object provides the APIs to send this type of asynchronous messages. Each message is also identified by a path in the same way as `DataItems`.

4. A `ChannelClient` offers an alternative set of API methods to send large files for media formats like music and video (in streaming as well) which save disk space over `Assets`. `ChannelClient` are also identified by a unique path.

The Wearable API also specify callbacks to listen for events that receive wearable data transmissions. Table 3.3 shows a summary of these objects and their corresponding callbacks. The 16 data types supported by `DataItems` can be found in the API documentation [125]. The full list contains a mix of Java types and Android-specific abstractions.

Data Type - Channel	Channel Type	Data Type	Listeners
`Messages - MessageClient`	Asynchronous/not-reliable	Bytes	OnMessageReceived
`DataItems - DataClient`	Synchronous/reliable	16 types	OnDataChanged
`Assets - DataClient`	Synchronous/reliable	Binaries	OnDataChanged
`Channel - ChannelClient`	Synchronous/reliable	Files	OnChannelOpened

Table 3.3: Map between the different data types and the available channels in the `Data Layer` API.

Data Layer Communication Flow

Once two devices are paired, the mobile and its companion apps can talk to each other through the `Data Layer` as long as they are signed with the same certificate. This is a restriction introduced for security reasons. Apps can use the `Data Layer` to open synchronous and asynchronous channels over the wireless channel. Table 3.3 shows the channel type corresponding to each abstraction of the `Data Layer`.

The `MessageClient` (asynchronous API) exposes the methods to put a message into a queue without checking if the message ever reaches its destination. This abstraction encapsulates the context of messages (destination and payload) into a single API invocation. Synchronous channels (`DataClient`, `ChannelClient`) provide transparent item synchronization across all devices connected to the network. These synchronous channels use many APIs to create the context of one transmission (in contrast to asynchronous). From now on, we will use synchronous channels to explain the operation of the `Data Layer` as these are more complex than asynchronous. This operation mode is important to understand how our tool `WearFlow` (Chapter 4) improves the tracking of mobile-wearable flows.

The context of one transmission consists of: the node identifier, channel type (see table 3.3 for the options), channel path (string identifier), and the data that will be transferred. Listing 3.2 shows the code for a synchronous `DataClient` example. The node identifier is a string representing a node in the Wear OS network (line 2). A channel path uses the identifier to generate a unique address which identifies each open channel within a node (line4). Finally, the data is the payload of the transmission (line 4).

```
1  public class MainActivity extends Activity {
2      private static final String COUNT_KEY = "com.example.key";
3      private DataClient dataClient;
4      private int payload = 0;
5
6      private void increaseCounter() {
7          // Create a data map and put payload in it
8          PutDataMapRequest putDataMapReq =
9          PutDataMapRequest.create("/count");
10         putDataMapReq.getDataMap().putInt(COUNT_KEY, payload++);
11         PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
12         // synchronise item
13         Task<DataItem> putDataTask = dataClient.putDataItem(putDataReq);
14     }
15 }
```

Listing 3.2: Data Client API example

An app can create many channels of the same type to send different payloads to the companion app. Developers often use path patterns to create a hierarchy that matches the project structure to identify different channels. For instance, the path *example.message.normal* can be used

to request a normal update, while the path *example.message.urgent* could indicate an urgent request.

To initiate a Mobile-Wear communication, the sender app needs to create the context of the channel through a sequence of APIs calls. Then, the Google Play Services app on the phone performs the transmission, handling the encapsulation, serialisation, and retransmission (if needed). In the smartwatch, Google Play Services receives the communication and processes the data before handing it over to the wearable app. The receiver app implements a listener that captures events from Google Play Services. The listener could be defined in a background service or an activity where the data is finally processed. We provide an extended example in Chapter 4.

3.3 Smart TV Platform

Android TV is the Android version for smart TVs. This OS version is heavily customised to support specialise hardware and system functionalities such as codecs and wide screen rendering. Many of these operations are protected by Android TV permissions such as HDMI_CEC and CONFIGURE_DISPLAY_COLOR_TRANSFORM. On the contrary, Android TV does not support many features available in other platforms because it is designed for a different purpose. Some of these features are touchscreen, telephony services, sensors, camera, among others.

TV apps offer an additional abstraction that enables the interaction with media content from the Internet and Smart TV hardware. TV apps require custom configurations in the APK Manifest to run on Android TV devices [126]: 1) The APK must not declare unsupported hardware such as a touchscreen. 2) It must declare a launcher TV activity. 3) It can optionally declare support of the `Leanback` library that provides user interface templates, paging, and other features that are exclusively for Android TV.

TV apps possess the same structure as mobile apps and use the same languages and development tools (described in the previous chapter). This allows developers to extend their mobile apps to support Android TV or create a new TV app from scratch. The most important difference with apps from other platforms resides in the user interaction. Users are expected to watch TV from medium distances, the input is based on a directional pad and a select button, and specialised hardware and codecs are used to render the application on Smart TVs. These specific features create a custom execution environment, e.g., permissions, libraries, and Android APIs.

Smart TV users interact with the apps through the home screen. The home screen (Figure 3.7) is the Android TV's main interface that provides access to apps, content recommendations, and global search. Users can access apps directly via the Apps menu or by searching channels or programs that apps add to the Home screen. Android TV also enables interaction with other devices. Figure 3.8 shows an example where an Android TV device is provisioned using a nearby Android phone. TV users can also use a remote-control app to engage with the TV app UI.

TV apps deliver media content or provide standard utilities. Like other Android apps, TV

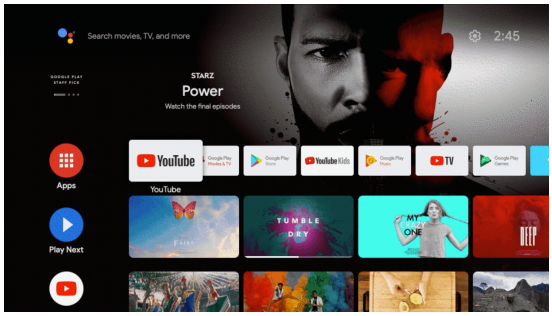


Figure 3.7: Android TV homescreen

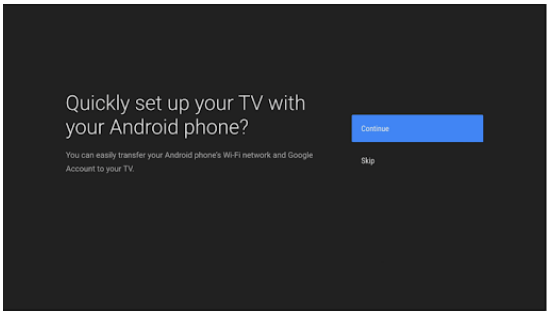


Figure 3.8: Android TV and mobile interaction

apps have access to the file system, sensors, and network connections. However, developers are restricted by the limitations of the underlying hardware, e.g., no precise geolocation due to the lack of GPS chipsets or telephony services. The TV Input Framework (TIF) is a set of libraries that facilitates the interaction of TV apps with media source providers [126]. Examples of TV apps implementing the TIF are Netflix, HBO, and Disney+. The TIF offers an interface to build apps emulating the TV broadcast style. It specifies channels, programs, track information, sessions, and other components required to display media content in a TV broadcast style.

Communication in TV apps

TV apps can establish connections with other hosts. For instance, a TV app can exchange data with a remote server or with a device connected to the same network or in close proximity. For this, developers can use plain sockets [127–129] or high-level Android APIs like the Nearby platform [54, 130].

The Nearby platform is available for proximity communication since Android 4.0. It provides the `NearbyConnection` and `NearbyMessage` APIs to communicate with nearby devices that are not required to be connected to the same network. The `NearbyConnection` API offers the capability to discover and connect with other devices using multiple protocols, while the `NearbyMessage` provides a publish/subscribe communication model. Similarly, the `WifiDirect` API allows two devices to communicate even if the two are not connected to the same network. The CastTV API allows mobile apps to display content on a Smart TV, but this interaction is handled internally by Android TV.

3.3.1 Mobile vs TV apps

We conducted a study of popular TV apps available in the Play Store similar to the preliminary study of Wear OS applications. For this, we downloaded 65 popular streaming APKs (as of January 2022). 75% of these APKs are pairs of mobile and TV apps, and the remaining 25% correspond to APKs compatible with both platforms. We call this dataset `popular-streaming`, and we further use it for the traffic analysis experiments in Chapter 5.

We use the same tools for this preliminary study, namely LibScout and LibRadar to analyse third party-libraries and Androguard for permissions. In Addition, we extracted metadata from

the Play Store, such as user reviews and the last updated date. Table 3.4 shows the number of downloads and user rating for the streaming apps. By analysing the Play Store metadata, we noted a significant degradation in the quality of the TV version of mobile apps. This degradation is evident when looking at users' ratings, reviews, and last app update from developers. For instance, most TV apps have a significantly lower user rating than the mobile version. Two examples are the YouTube apps (mobile version 4.3 vs TV 2.2) and CNN apps (mobile 4.5 vs TV 1.3). A manual verification of Play Store reviews is consistent with the rating as users constantly complain about buggy and unusable apps.

We also note that TV app updates come with a delay compared to the mobile version, between one month and one year, with 50% of the TV apps taking at least three months. Worst, two TV apps have been left without updates since 2018 (`com.playstation.video.atv` and 2019 `com.ted.android.tv`). These apps are still available in the PlayStore, and their mobile versions are up to date (2022). These issues show that developers give less attention to their TV apps. This might be because of the much bigger user base in the mobile ecosystem (see Table 3.4). However, this leaves TV users vulnerable to bugs and bad development practices. We show that the difference between TV and mobile apps updates is more significant for less popular apps in Chapter 5.

The preliminary study shows that TV apps use fewer permissions than mobile apps. We found 27 unique AOSP and 20 third-party permissions requested by TV apps. In contrast, we found 42 AOSP and 76 third-party permissions in mobile apps. However, we detected 18 permissions requested exclusively in TV apps. For instance, permissions to read and write Electronic Program Guide (EPG) data, digital rights (DRM), and integration with the Amazon Fire TV Launcher. Third-party libraries follow a similar pattern. In general, mobile apps embed more TPL than TV apps, but there are libraries found exclusively in TV apps. There are two exceptions in this dataset of popular streaming apps with more libraries in the TV version (the Curiosity Stream and WWE apps). We give an in-depth analysis of the Android TV ecosystem in Chapter 3.1, so we leave the details for that chapter.

3.4 Android Platforms Threats

Android smart devices store private data, including PII (e.g., device identifiers) and other sensitive data such as biomedical data, sensors, and multimedia. All this data available open the doors for unintentional or malicious data leaks and attacks exploiting vulnerabilities. The research community develop frameworks to address these threats. We refer the reader to Section 2.3 for an overview of static and dynamic analysis frameworks for Android apps.

Unfortunately, most of these frameworks focus on mobile apps and do not consider the specific features of other platforms. For instance, Liu *et al.* conducted a large-scale comparative study between mobile and TV apps [131]. They showed that TV and mobile versions of the same app often encounter different kinds of security vulnerabilities. Aafer *et al.* discovered 37 unique vulnerabilities by analysing the firmware of 11 Android TV boxes. The nature Wear OS and Android TV enforces developer to adapt their apps to the resources available on each platform.

App Name	Downloads-Mob	Downloads-TV	Rating-Mob	Rating-TV
Netflix	1B	50M	4.3	2.4
Amazon Prime Video	100M	50M	4.3	1.6
Disney+	100M	-	4.4	-
Twitch	100M	-	4.6	-
Youtube	10B	100M	4.3	2.2
Pluto TV	50M	-	4.1	-
Plex	10M	-	4.1	-
Hulu	50M	5M	3.9	1.7
Sling TV	10M	-	4.1	-
BBC	10M	100K	4.1	2
Natgeo TV	1M	-	4.1	-
Tik tok	1B	100k	4.4	*
ESPN	50M	-	4.1	-
AppleTV	1M	-	2.3	-
RedBull TV	10M	-	4.3	-
WWE	10M	100k	4.5	3.8
Old Movies	5M	-	4.5	-
Rakuten TV	1M	1M	4	1.7
Kodi	10M	-	3.8	-
HBO GO	5M	500K	4.3	1.6
CNN	50M	100K	4.5	1.3
haystack	1M	-	4.3	-
curiosity stream	1M	100K	4	3.7
TED talks	10M	5M	4.5	4
Lifetime A&E	5M	-	4	-
MasterClass	1M	50k	4.7	*
Play Station	100M	100k	3.8	1.2
NBC sports	5M	100k	3.4	1.5
NFL	100M	-	4.3	-
moviestar	10M	500k	4.2	2
ITV hub	10M	100M	3.7	-
Stadia	1M	100K	3.8	4.1
WRC	500K	10K	3	2.3
earthcam	1M	50k	3.2	3.6
CBN family	100k	10k	4.4	*
Yupp TV	10M	1M	3.5	2.5
Eros Now	10M	1M	2.8	2.7
Jellyfin	100K	100k	3.9	3.8
Player	10M	500k	3.7	2.3
Acorn TV	100k	50k	3.3	1.5

Table 3.4: Comparison between Mobile and TV versions of Streaming Apps. (-) indicates only one version and (*) indicates information not available

This adaption in conjunction with bad development practices and the lack of proper guidance from Google, results in new threats that are platform-specific.

Although the risk of privacy and security problems is present across all platforms, the modes in which the threats occur vary. This led us to pose new questions regarding user security and privacy. Thus, we continue this thesis by analysing the threats in Wear OS and Android TV apps. We first discuss these threats and then study some of these issues in the following chapters.

Wear OS Threats. We are primarily interested in understanding wearable channels that can lead to sensitive data leaks. The `Data Layer` (described in Section 3.2.2) enables inter-device channels where sensitive data can flow across devices before being leaked. Using this channel, a wearable app could read sensor data and share with its mobile counterpart, which finally leaks the data. This situation expands the context of wear/mobile applications across more than one device. Therefore, we cannot assess the security of an Android app by just looking at the mobile or wearable version in a vacuum. Instead, we need to consider both apps as part of the same execution context.

Wear OS does not support many of the traditional mechanisms for inter-device communication, e.g., sockets and Bluetooth, making the Data Layer the unique channel for Mobile-Wear communications, and its study fundamental to understand information flows in Wear OS. None of the available frameworks model this interaction properly (see Section 4.7 for related work). Thus, in Chapter 4 we propose `WearFlow`, a static analysis framework that models the Mobile-Wear communication. `WearFlow` enables the tracking of sensitive data across mobile and wearable apps.

While `WearFlow` addresses this specific feature of Wear OS applications, it is important to highlight that Mobile-Wear communication is just one out of many problems specific to this platform. Sikder *et al.* published a survey on sensor-based threats and attacks to smart devices and applications [132]. They describe many attacks using sensors to capture sensitive information, including wearable apps. Wang *et al.* study keyboard attack inferences using smartwatches sensors [7]. A study of Ching *et al.* presents evidence that low computing power cause developers to avoid strong security mechanisms in smartwatches [133]. An industry report describes authentication and secure connection vulnerabilities in smartwatches [6]. Goyal *et al.* study the security of wearable health trackers, and they found that wearables are susceptible to multiple attacks, including DoS, hardware deactivation and traffic interception [8].

All these problems support the argument that the wearable ecosystem needs to be studied appropriately, and we take care of one aspect of this platform in the next chapter. The specific threats we consider for the wearable ecosystem are stated in section 4.1.2.

Android TV Threats. Similarly to the wearable ecosystem, we are interested in security and privacy threats for TV apps. However, the context of this platform determines different threats. Android TV primary purpose is to deliver media content. Users' viewing history and preferences are exposed to trackers. Most users are unaware of profiling risks or consider advertisers accessing

their viewing history unacceptable [30]. Thus, we consider the threats pose by data leaks and the relation of these events with third-party libraries and tracker domains.

TV apps rely on APIs such as sockets or the Nearby platform for inter-device communication (the `Data Layer` is not available). These APIs allow developers more freedom but also increase the risk of insecure communications. Thus, we look at how TV app developers implement inter-device communication and their privacy implications. Another threat for TV apps resides in bad development practices due to porting mobile apps to Android TV. For instance, inconsistent permissions and duplicated permissions. The specific threats considered for the Android TV evaluation are stated in Section 5.1.1.

Note that we do not consider the same problems on both platforms. For instance, we do not perform a large-scale study of permissions and third-party issues for wearable apps, even though these might exist. We propose extensions to our work in Section 7.2.

3.5 Chapter Summary

Abstractions in Android vary across platforms because of hardware limitations and platform design. These differences result in syntactic and semantic differences that security analysts must consider as it affects the study of information flows across platforms RQ1. Thus, analysing multi-platform apps requires the following considerations.

- Specific APIs might result in complex implementation patterns that need to be modelled properly for precise analyses, e.g., the `Data Layer`. Additionally, different stakeholders offer third-party libraries that are specific to each platform. Thus, the collection of library signatures needs to be complemented.
- Taint specifications will differ for each platform as the available APIs differ. Unfortunately, this situation yields an overload of manual analysis to generate specifications or requires precise automated tools.
- Experiments must consider hardware limitations and how users interact with apps on each platform.

We argue that unique characteristics require custom analyses for each platform RQ2. At the same time, there are common tasks that could be automated RQ3, such as generating taint specifications. Chapter 4 presents a framework that enables inter-device tracking in the wearable platform RQ1 and a new benchmark suite designed for testing. Chapter 5 studies information flows in TV apps using static/dynamic analysis RQ1. This chapter also includes a study of platform-specific characteristics, such as the Android TV stakeholders and architecture RQ2. Last, in Chapter 6 we give the first steps towards a general framework RQ3 by generating taint specifications for arbitrary platforms using NLP and software documentation.

4 Enabling Information Flow Analysis in Wear OS

In this chapter we present WearFlow, a static analysis tool that enables information flows analysis in the Mobile-Wear ecosystem. We evaluate WearFlow on a new benchmark suite WearBench and search evidence of potential information leaks on a real-world dataset of mobile and wearable apps.

4.1 Introduction

After exploring the similarities and differences of the multiple Android ecosystem in chapter 3, we start by analysing the limitations of current frameworks to evaluate apps in the wearable ecosystem.

Wearable companion apps create a richer execution environment that expands the context of mobile applications to Wear OS devices. Although beneficial for users, this expanded environment introduces new security and privacy issues [6–9]. One such issue is that current information flow analysis techniques cannot capture the communication between devices with precision (details in Section 4.7). This limitation can lead to undetected privacy leaks when developers use these channels.

The security community leverages information flow analysis to systematically study how apps use sensitive data. Most works limit the scope of their analysis to single-app execution contexts [32–37]. Frameworks such as COVERT [43], DidFail [36], and DialDroid [39] augment the scope of the data tracking to include inter-app data flows which use inter-component communication (ICC) methods. These works expand the execution context from one mobile app to a set of mobile apps.

In contrast to previous problems, information flow analysis in the wearable ecosystem needs to track sensitive data across apps in different devices, i.e.: the handheld and the wearable. In other words, it needs to consider that data flows can propagate from the mobile app to its companion app through the wireless connection. We describe these flows as mobile-wear for simplicity, but the same principles apply to wear-mobile flows. As we have seen in the previous chapter, in Android and Wear OS, this low-level communication is managed by Google Play Services (Data Layer), whilst apps embed the APIs to interface with this service.

A static analyser would need to propagate data flows across the mobile app, the Google Play Services app, and the wearable app to detect inter-device leaks. Our goal is to track sensitive

data across this communication without propagating data flows through Google Play Services code. Google Play Services is not open source, and most of its implementation is in native code. Even with the code available, this intermediate step is constant, which means a waste of resources and time when doing the data flow analysis.

We described the communication flow of the Data Layer in Section 3.2.3. These flows are not one simple API call, but require multiple API calls to generate the context and then perform the communication. We solve the inter-device tracking problem by creating a model of the Data Layer based on its public APIs. Then, we use this model to reason about the communication between mobile and wearable apps with the aim of capturing inter-device flows. This allows us to run taint analysis in an extended context that comprises mobile and companion apps.

In summary, we this chapter makes the following contributions:

- We propose `WearFlow`, an open-source tool that uses a set of program analysis techniques to track sensitive data flows across mobile and wearable companion apps. `WearFlow` includes library modelling, obfuscation-resilient APIs identification, string value analysis, and inter-device data tracking.
- We develop `WearBench`, a novel benchmark to analyse Mobile-Wear communications. This test suite contains examples of mobile and wearable apps sharing and exfiltrating sensitive data using wearable APIs as the communication channel.
- We conduct a large scale analysis of real-world apps. Our analysis reveals that real-world apps use wearable APIs to send sensitive information across devices. Our findings show that developers are not using data sharing APIs following the guidelines given by Google.

4.1.1 Motivation Example

The following code snippets show an example of a data leak using the `DataItem` channel. Here, a wearable app sends sensitive information to the Internet after a mobile app transfers this data through a synchronous communication offered by the `DataClient` API. Note that this is a simplified code instead of the full Java implementation (as shown in Listing 3.2).

In Listing 4.1, the mobile app sends the geolocation and a constant string to the companion wearable app. First, the channel is created (line 4) with its corresponding path. Then the geolocation and a string “hello” are added to the channel in line 5, and 6 respectively. Finally, the app synchronises all the aggregated data in one API call (`synchronizeData`) in line 7.

```
1 nodeID = getSmartwatchId()
2 location = getGeolocation()      // source
3 text = "Hello"
4 channel = WearAPI.createChannel("path_x")  // channel path
5 WearAPI.put(channel, "sensitive", location)
6 WearAPI.put(channel, "non-sensitive", text)
7 WearAPI.synchronizeData(nodeID, channel)    // communication API
```

Listing 4.1: Simplified example of mobile app sending data to the companion app

The wearable app receives this transmission (Listing 4.2) and after parsing the request, it sends the location and constant string to the Internet, although only the location is sensitive. First, the app fetches the event from the channel using the `Data Layer` API (line 1). The developer uses a conditional statement to execute actions depending on the event's path. Paths are the only way to characterise events that trigger different data processing strategies when exchanging data through a channel. Here, *path_x* (lines 3 to 7) corresponds to the branch that handles the data sent by listing 4.1, which includes the geolocation. In this case, the geolocation is sent via a sink in the companion app to the Internet. The branch *path_y* is used to process a different event not relevant in our example.

```
1 event = WearAPI.getSynchronizationEvent()
2 if (event.path == "path_x"){       // channel path
3     data = parseEvent(event)
4     location = data["sensitive"]
5     hello = data["non-sensitive"]
6     exposeToInternet(location)    // sink 1, leak
7     exposeToInternet(hello)       // sink 2, no leak
8 }else if (event.path == "path_y"){
9     do_something_else
10 }
```

Listing 4.2: Example of companion app exfiltrating sensitive data.

On the receiver side, it is also possible to specify the channel path in the Manifest using an Intent-filter. In this case, the service only receives events that match the path specified in the Intent-filter. It is also possible to specify a path prefix, instead of the full path, and then trigger different branches in the code. For Listener defined in Activities, developers rely on indirect references to the path on the code, like the example.

4.1.2 Threat Model

We identify the following security risks that arise from the transmission of PII in the mobile-wear ecosystem:

1. Re-delegation: The permission model in Android requires developers to declare the permissions of their mobile and wearable apps separately. This enables mobile and wearable apps to engage in colluding behaviours [48]. For instance, a mobile app that requests the

`READ\_CONTACTS` permission, can use the `Data Layer` APIs to send the contact information to a wearable application that does not have this permission. Similarly, a wearable application could share sensor data such as heart rate or other sensors with its corresponding mobile app without requiring access to Google Fit permissions.

2. Wearable data leaks: Wear OS includes APIs to perform HTTP and other network requests to Internet-facing services. This means that wearable apps have exactly the same capabilities to exfiltrate data as regular mobile apps. However, as already mentioned in Section 4.1, information flow tools available today only account for data leaks that happen directly via the mobile app (or via other apps in the case of collusion). As of today, there are no methods to detect data leaks through wearable interfaces.

3. Mobile data leaks: In a similar way, the mobile app could exfiltrate sensitive data leaked from the wearable app environment. An example of sensitive data unique to the wearable is the heart rate. A mobile application can pull this data and send it over the network. Note that while this threat can be materialised through a permission re-delegation attack, it is not strictly bound to this attack. Instead, both apps can request permission to access specific sensitive data, but the taint is lost when data is transmitted from the companion to the mobile app.

4. Layout obfuscation: Developers are increasingly using obfuscation techniques to prevent reverse engineering and to shrink the size of their apps [42]. Obfuscation presents a challenge to information flow analysis when it modifies the signature of relevant classes and methods. In our case, the APIs from the `Data Layer` might be obfuscated, and we cannot merely look at the signatures of the API methods.

4.2 Modelling Mobile-Wear Communication

Note that a Mobile-Wear taint tracking needs to consider that data flows are combined in a single point when the sender transmits the `DataItem`, and it separates again when the receiver app parses the event. This is shown in Listing 4.1, where the `Data Layer` aggregates data (i.e., the geolocation and a constant value) into a single channel. Finally, we note that an attacker may use any other channel described in Section 3.2.3 to leak sensitive data, although the technical procedure will be different.

Mobile-Wear taint tracking presents a different set of constraints and characteristics than Inter-Application and Inter-Component communication analysis. In Wear OS, the communication between the smartphone and the wearable involves the mobile app, Google Play Services, and the wearable app. As the Google Play Services library acts as a bridge between the two, we need to model its behaviour to track information between the two apps.

As seen in the examples shown in Listings 4.1 and 4.2, wearable APIs are designed to send and receive data in batches. This means that developers first insert the different items they want to transmit between apps and then execute a synchronization API call. From an information

flow analysis perspective, this means that multiple data flows join into a single point when an app invokes the synchronization API to send data. One possible solution would be to taint all the information exchanged. However, this overestimation would result in a high number of false positives. There is another challenge behind tracking individual data flows in Wear OS, i.e.: Google Play Services is not open source and it is implemented in native code, which makes the data tracking more difficult [45].

In order to track these flows, we have created a model of the `Data Layer` to generate a custom implementation of the wearable client library. To create the model, we manually inspected the wearable-APIs from the `Data Layer`, and built a sequence of possible invocations and the effect of these APIs on the context of the communication. This model allows us to extract the context of each communication, such as the *path* and the data added into a *channel*. Then, we can use this information to replace the invocations to the original APIs with invocations to our instrumented APIs.

Note that we do not know the details of how Google Play Services implements the communication, but we do know the result of the communication, and we can reason about the context of communication by looking at relevant points where the apps invoke wearable APIs.

The result of our model is a mapping between the original methods from the `Data Layer` client library to a modified implementation template that facilitates the matching of individual data flows between apps. This modified implementation is generated as follows:

1. We identify all relevant classes from the gms-wearable library and generate custom signatures for each method.

2. For each app, we identify all invocations of synchronous and asynchronous APIs from the `Data Layer`. For each invocation, we run slice and taint analysis to extract the context of the transmission. This involves:

   (a) Identifying the channel creation.

   (b) Searching the items that have been added to the channel variable (data sent across the channel).

   (c) Evaluating strings from the context (path and keys).

   (d) Generating custom API calls using the extracted context and corresponding method template.

   (e) Replacing original method invocation with a custom API invocation.

By doing this, we can simulate the propagation of data flows across apps on different devices while keeping the semantics of the different data flows intact. As an example, whenever we find a call to `<DataClient: putDataItem(PutDataRequest request)>`, the model will tell us that this is a synchronous communication which in sending a DataItem (encapsulated in the

`PutDataRequest`). In this case, the model also specified that the `PutDataRequest` object required a previous API call that creates a channel, and other APIs that add data to the `DataItem`. We use this information to do a backward and forward inter-procedural analysis to extract such information. Finally, the model provides the rules to match entry and exit points once we have the results of the data flow analysis.

To the best of our knowledge, this is the first framework that models Mobile-Wear communication. As a consequence, current frameworks fail to detect the situations above. This happens either when developers are not following good coding practices or when miscreants intentionally try to evade detection mechanisms that rely on data-flow analysis.

4.3 Implementation: WearFlow

We design a pipeline of five phases that result in the detection of Mobile-Wear data leaks. Figure 4.1 shows a high-level overview of our system. Phase 1 converts the app to a convenient representation and extracts relevant information. Phase 2 deobfuscates (if necessary) the Google Play Services client library and relevant app components. Phase 3 performs the context extraction and instrumentation for every invocation to a wearable API. Phase 4, runs the information flow analysis and exports the results. Finally, we match data flows according to the model of the `Data Layer` in phase 5 to obtain all Mobile-Wear flows.

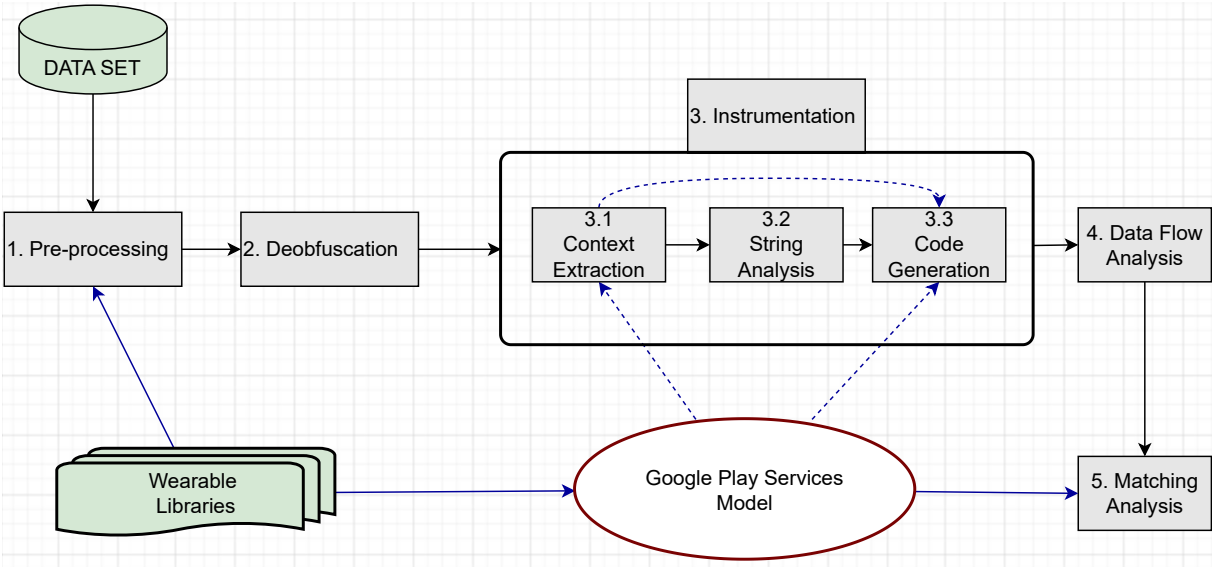


Figure 4.1: Overview of WearFlow.

Phase 1: Pre-Processing

Android packs together the wearable and the mobile app into a single package file (namely, APK). WearFlow first splits both apps and then uses Soot to pre-process each executable separately. In particular, we convert the Dalvik bytecode into the Jimple Intermediate Representation (IR), and parse the relevant configuration files (e.g., the wearable and mobile app Manifests). Jimple simplifies the different program analysis techniques we use in the following phases.

WearFlow then searches for Wear OS components (services and callbacks as in Table 3.3), subject to an optional deobfuscation phase (Phase 2). We leverage the Manifests to understand the relationship between paths and services by looking at the intent filters declared as *WearableService*. We then inspect the Jimple to obtain all variables of the data types listed in Table 3.3, including those that appear in callbacks. These data types are used to open Mobile-Wear channels. We will instrument all these components as described in Phase 3.

Phase 2: Deobfuscation

We use a simple heuristic to detect if the app is obfuscated. First, we assume that all Mobile-Wear applications would use any of the methods from the classes of the `Data Layer` API shown in Table 3.3. Thus, we search for these methods in the client libraries of the APK. If no method is found, we consider the app may be obfuscated and perform a type signature brute-force search. This signature models the type of inputs and outputs of a function.

In addition to the type signature, we further look at local variables declared using system types in the method and compute their frequency per method (when the method is not a stub). The rationale behind including context from the method itself is to reduce the number of false positives when performing the signature search. The signature model uses only system types and abstract types from the `Data Layer` to generate the obfuscation-resilient signatures. We refer the reader to Section 4.6 for a discussion on our choices and how this may impact our results.

We extracted signatures for all relevant methods that model Mobile-Wear IPC (see Section 3.2.3). Overall, we produced 63 signatures of methods that exchange `Messages`, `DataItems`, `Assets` and `Channels`. We then search for methods in the app's components that match against these signatures. When we find a match, we identify the corresponding wearable API of our interest. As we show in Section 4.5.2, we can identify all the methods used by the model with the above features.

Phase 3: Instrumentation

This phase aims at instrumenting the apps under analysis, and it has three steps: context extraction, string analysis, and code generation. It takes as input a model of the Google Play Services library. Our attempt to model this library is described in Section 4.2.

For the context extraction, we search invocations to wearable APIs that send or receive data in each of the components seen in the pre-processing step. Once that one API is identified, WearFlow performs an inter-procedural backward analysis to find the creation of the corresponding channel and then a forward analysis to find invocations to APIs which add data to the channel. We then evaluate strings of relevant API methods; for instance, the method `<PutDataMapRequest.create(String path)>` for `DataItems`. For this, we perform an inter-procedural and context-sensitive string analysis. For asynchronous APIs (e.g., `MessageClient`), the context extraction is limited to evaluate the variable which contains the channel path.

The next step is to instrument the app. On the one hand, we add our custom methods to the

client library. In particular, we add the declaration of the method we use as entry/exit points in their corresponding classes. On the other hand, for each invocation to APIs methods acting as entry/exit points, we generate the corresponding invocation to our custom APIs. We use the output of the context extraction, string analysis, and the model of the `Data Layer` to generate such invocations. The resulting code will replace the original invocations.

The Listing 4.3 shows the instrumented code corresponding to the motivation example in Listing 4.1. The lines [8,9] replace the commented line. Note that the code is simplified for readability. A more realistic example of a synchronous transmission can be found in Listing 3.2. Moreover, our goal with this example is to give a notion of the instrumentation, which is done in the Jimple IR, not in Java.

```
1 nodeID = getSmartwatchId()
2 text = "hello"
3 location = getGeolocation()
4 channel = WearAPI.createChannel("path_x")
5 WearAPI.put(channel, "sensitive", location)
6 WearAPI.put(channel, "non-sensitive", text)
7 // WearAPI.synchronizeData(nodeID, channel) -- replaced API call
8 WearAPI.syncString(nodeID, channel, "non-sensitive", text).
9 WearAPI.syncString(nodeID, channel, "sensitive, location)
```

Listing 4.3: Simplified instrumented code.

Phase 4: Data Flow Analysis

This phase performs data flow analysis of the Mobile-Wear ecosystem as a whole. First, we add callbacks from the Wear OS libraries as given by our model to enable data flow analysis across devices (see Section 3.2.3) . Additionally, we add data wrappers that can capture how data flows propagate through objects of the `Data Layer`. The next step is to compute the call graph of both apps and perform a taint tracking analysis as a single context. We do this by first running the taint analysis separately on each app and then matching the results using the instrumented APIs as connectors between data flows. We add the custom APIs that send data as sinks (wearable-sinks) and the custom APIs that receive data as sources (wearable-sources). Then, we add the wearable-sources and wearable-sinks to the list of sources and sinks.

Finally, this phase reports the results of the taint tracking. At this point, we are only interested in data flows with wearable-sources or wearable-sinks. It is worth noting that taint analysis still detects data flows that end in a non-wearable sink, but they are irrelevant to the matching step. Our approach is agnostic to the underlying method used to compute data flows. We refer to Section 4.3.1 for implementation details.

Phase 5: Matching Analysis

The final step consists of matching exit points with entry points; that is to say, wearable-sinks with wearable-sources. We consider three values to match data flows: channel path, API method,

and key. If the value of the path or key could not be calculated during the context extraction, then we use a wildcard value that matches any value. To match the API methods, we built a semantic table that provides information to match wearable-sinks with their corresponding wearable-sources. We present a summary in Table 4.1 for readability. The table contains thirty-four entries that can be found in the project repository.

Library	Wearable sink signature	Wearable source signature
DataClient	void putString(String,String)	DataMap: String getString(String)
MessageClient	Task sendMessage(String,String,byte[])	MessageEvent: byte[] getData()
DataClient	void putAsset(String,Asset)	DataMap: Asset getAsset(String)
ChannelClient	sendFile(Channel,Uri)	Task receiveFile(Channel,Uri,Bool,String)

Table 4.1: Selection of Sink-Source matches in Data Layer API. A full list can be found in WearFlow repository.

4.3.1 Implementation

WearFlow relies on the Soot framework [80] to perform the de-obfuscation, context extraction and app instrumentation (Phases 2, 3.1 and 3.3). Our implementation leverages FlowDroid [32] with a timeout of 8 minutes per app for the information flow analysis (Phase 4) and Violist [134] for the string analysis (Phase 3.2). We use FlowDroid and Soot because previous works report that they provide a good balance between accuracy and performance on real-world apps, as mentioned in Chapter 2. We customize FlowDroid to run on wearable apps by adding callbacks from the Wear OS libraries and by extending the SuSi [51] sources and sinks, as discussed previously. We also performed several optimisations to Violist to reduce the execution time while keeping the accuracy for the APIs we were interested in. For instance, we reuse the control flow graph generated by Soot, and we limit the evaluation of the strings to relevant methods. With this, WearFlow adds around 6000 LoC to these frameworks. The link to the repository can be found in Section 1.4.

4.4 Benchmark suite: WearBench

To evaluate the precision and recall of `WearFlow` against other tools, we first need test cases and their corresponding ground truth, e.g., an app dataset and the list of data flows exfiltrating sensitive data. Note that while real-world apps can be useful to perform a large-scale analysis of potential data leaks, they are not practical to evaluate tools because getting the ground truth for each app is unfeasible. Therefore, we require a test suit of apps that comes with complete data flow results beforehand.

As the community lacks a test suite that includes Mobile-Wear information flows for Android, we created `WearBench`. WearBench comprises 15 pairs of Mobile-Wear Android apps with 23 information flows between the mobile app and the wearable companion (18 sensitives). Our test suite covers examples of all APIs from the `Data Layer`. It is designed to test different scenarios on the sender and receiver side of the communication, e.g., the mobile app sends sensitive and non-sensitive data to the wearable app. If both are leaked on the wearable app, only the information flow with the sensitive data should be reported. All apps were tested on actual devices to confirm

the runtime behaviour. The link to the repository can be found in Section 1.4.

Our suite is inspired by `Droid-Bench`[1] and `ICC-Bench`[2], which are standard benchmarks to evaluate data flow tools. Note that these benchmarks evaluate the effectiveness of the taint analysis and some Inter-App communication cases using ICC methods. Instead, we are evaluating Inter-App communication between mobile and wearable apps (using the `Data Layer` API). Therefore, we cannot use these benchmarks alone to evaluate `WearFlow` or any other tool that searches sensitive information flows in the Mobile-Wear ecosystem.

WearBench Apps Description

We provide a brief description of the test cases grouped by API type and an expanded description of each test case in the Appendix.

Message Client. WearBench contains 5 test cases where the mobile app sends sensitive data to the wearable app using the `Message Client` API. We implemented test cases using APIs for Wear OS 1 and Wear OS 2. Our test cases cover situations where the app receives messages in Services and Activities. Apps that receive messages in the main thread (Activities) must implement lifecycle methods to capture the message event, while apps receiving messages in Services must overwrite the *onMessageReceive* callback. The main challenge for `Message Client` APIs is matching individual data flows using the correct path on the receiver side.

Data Client. We developed 9 test cases for the `DataClient`. There are more apps for this API because of its mode of interaction. There are multiple ways to construct the payload of the synchronisation events, and we made sure to cover all of them. For instance, the `DataMap` API offers methods to add primitive type objects (string, float) to the context of the communication or add more abstractions that group several objects. Similarly to `Message Client` cases, we implemented apps using the API version for Wear OS 1 and Wear OS 2. Static analysers must distinguish different channels and data in the payload.

Channel Client. This API is similar to the `Message Client` API. Therefore, we implemented one test case for this API. The main difference is the payload, as `Channel Client` objects support sending large files and streaming data.

4.5 Evaluation and Results

In this section, we report the evaluation of `WearFlow` using the APKs from `WearBench`. In our evaluation, we compare our results against FlowDroid. For this, we add the Data Layer APIs as sources and sinks, execute FlowDroid on both the mobile and wearable companion and look for matches. We run FlowDroid with a context-sensitive algorithm twice: first with high precision, we set the access path length to 3. Then we reduce the precision by setting the access path to 1. With this, FlowDroid truncates taints at level 1.

[1] https://github.com/secure-software-engineering/DroidBench
[2] https://github.com/fgwei/ICC-Bench

Table 4.2 shows the result grouped by API of our evaluation. `WearFlow` detects all 18 exfiltration attempts with two false positives. These two false positives stem from a branching sensitivity issue, which results in false positives during the data flow analysis (Phase 4). Conversely, FlowDroid with high precision (access path = 3) detects 7 out of 18 exfiltrations – these are matches communicating with the `Message Client` and `Channel Client` APIs, but it missed 13 sensitive flows. The FN are because FlowDroid fails to propagate taints on complex objects from the `Data Layer` with access path 3.

When reducing the precision (access path = 1), FlowDroid identifies the missing sensitive flows but at the cost of producing 12 false positives. This results from an overestimation of taints that uses `Data Client` APIs. With access path 1, FlowDroid cannot distinguish between different variables in the `Data Client` abstractions. Hence, the high number of FP. Table 4.3 provides the result for each test case in `WearBench`.

	Existing Data Flows			Found Data Flows		
Library	**Apps**	**Total**	**Sensitive**	**WearFlow**	**Flowdroid-HP**	**Flowdroid-LP**
DataItem	9	16	13	14 (1 FP, 1 FN)	0 (13 FN)	22 (6 FP)
Message	5	6	4	5 (1 FP)	6	10 (6 FP)
Channel	1	1	1	1	1	1

Table 4.2: Evaluation for WearFlow VS FlowDroid results on WearBench. HP = high precision, LP = low precision.

Our results show that `WearFlow` performs better than FlowDroid by a clear margin. This exemplifies how the modelling, instrumentation and matching analysis can improve information flow analysis in wearable applications.

4.5.1 Analysis of Real-World Apps

We use `WearFlow` to search the presence of potential data leaks on around 3.1K real-world APKs available in the Google Play Store (downloaded from AndroZoo [135]). From an initial set of 8K APKs, around 5K refer to standalone (only wear) APKs, and 3.1K include mobile and wearable components. We execute `WearFlow` against this set which corresponds to 220 different package names. Table 4.4 summarises the sensitive flows found in the real-world app dataset. Note that the dataset contains multiple versions of the same app. Thus, we refer to apps as APKs with unique package names. Table 4.5 show the wearable sources and sinks used to transmit data in the APKs. Note that there are significantly more entry points for `DataClient` sinks. This result suggests bulk transmissions when using this API.

Figure 4.2 shows a summary of the different APIs used as exit/entry points of sensitive data flows. Although we found the occurrence of the `ChannelClient` API in the dataset, we did not find any case where this API was used to send sensitive information. `WearFlow` identifies sensitive information flows that include the transmission of device contacts (via Cursor objects), location, activities, and HTTP traffic. We also found that sensitive data ended up in the device logs (17% of overall sinks) or SharedPreference files (20%). A more detailed analysis of these flows for a selection of apps is provided in Section 4.5.3.

	Existing Data Flows		Found Data Flows		
Test	Total	Sensitive	WearFlow	Flowdroid-HP	Flowdroid-LP
SimpleDataItem1	2	1	1	0	2
SimpleDataItem2	2	1	1	0	2
SimpleDataItem3	2	2	2	0	4
SimpleDataItem4	1	1	1	0	1
DataItem1	1	1	1	0	1
DataItem2	1	1	1	0	1
DataItemAssets	2	2	3	0	6
InterDataItem1	2	2	2	0	2
InterDataItem2	3	2	2	0	3
SimpleMessage1	1	1	1	1	1
SimpleMessage2	2	1	2	2	6
SimpleMessage3	1	1	0	1	1
SimpleMessage1	1	0	1	1	1
MessageClient1	1	1	1	1	1
SimpleChannel1	1	1	1	1	1

Table 4.3: Evaluation results for our test-suite between WearFlow and FlowDroid. HP = high precision, LP = low precision

	Apps	**APKs**
Number	220	3,111
With flows	47	293
With sensitive flows	6	82
Total flows	5,096	
Total sensitive flows	543	

Table 4.4: Summary of results.

wearable sinks	Exit Points	wearable sources	Entry Points
MessageClient	158	MessageEvent	148
DataClient	127	DataMap	534
ChannelClient	0	ChannelEvent	0

Table 4.5: Sources and Sinks types

Overall, `WearFlow` found 4,896 relevant data flows in all the analysed APKs. Out of those, 388 relate to Mobile-Wear sensitive information flows in 6 apps (or 50 APKs, when considering all versions and platforms). The results indicate that 70% of the flows are from the mobile to the wear platform, while 30% are wear to mobile.

4.5.2 Applicability

We next see how we perform when dealing with obfuscation and what is the runtime overhead.

Obfuscation vs Non-Obfuscation. `WearFlow` runs the deobfuscation module and performs the signature-based search if the wearable APIs are not found in the pre-processing phase. We consider the deobfuscation successful if we can reverse the API signatures of all relevant methods from the `Data Layer` to their original form. `WearFlow` detected 282 obfuscated APKs in the dataset. The deobfuscation phase successfully unmangles all these APKs. We found 71 data flows using the `Data Layer` within these APKs. `WearFlow` did not find relevant APIs in 651 APKs. This can be either because these APIs are not used or because developers use more complex obfuscation techniques. We discuss this limitation and alternative approaches in Section 4.6.

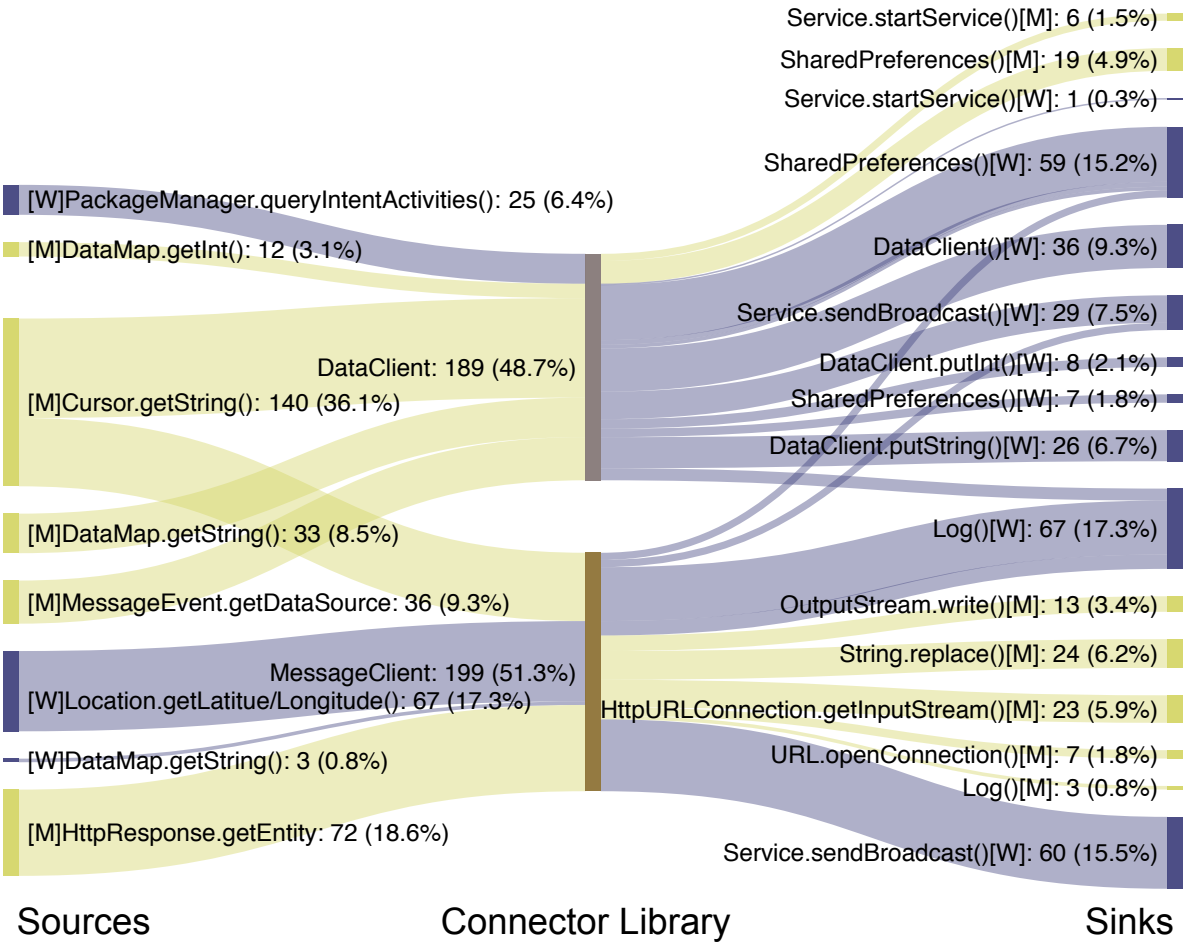


Figure 4.2: Sensitive information flows found. [M] refers to Android and [W] refers to Wear OS.

On the other hand, we found around 2K non-obfuscated APKs in our dataset (the `Data Layer` signatures were found without obfuscation). `WearFlow` instruments 4.8 components on average per APK (excluding library classes). From all the wearable APIs, around 48% are `DataClient` APIs, 51% —MessageClient— APIs, and less than 1% `ChannelClient` APIs. These numbers confirm that developers are aggregating multiple variables into a `DataClient` object before synchronising `DataItems`. The benefits of instrumenting the APKs to track individual data flows are clear in this scenario. FlowDroid either misses these flows or overestimates to impractical levels, as shown in the previous section.

Running time. Running our tool on the real-world dataset took 115 hours. `WearFlow` analyses over 95% of the APKs before the 8 minutes timeout lapses. The average time per APK is 3.1 minutes. Note that wearable apps are considerably smaller in size than mobile apps, and `WearFlow` evaluates most wearable apps in less than 1 minute. The time distribution per phase analysis is the following: pre-processing 13%, string analysis 9%, deobfuscation and instrumentation 2%, and data flow analysis 76%.

`WearFlow` failed to complete the analysis for a small number of APKs. In most cases, this is due to unexpected bytecode that Soot fails to handle, errors while parsing APK resources, or

because the analysis reached an extended timeout.

Overall, `WearFlow` detects data flows for an additional 282 Mobile-Wear APKs. Without the deobfuscation phase, these flows would not otherwise be extracted. Moreover, the deobfuscation phase only takes 2% of the running time.

4.5.3 Case Studies

This section describes issues found by `WearFlow` in specific apps in relation to the threat model presented in Section 4.1.2.

Companion Leak. We first study the case of Wego (*com.wego.android*), a travel app to book flights and hotels with more than 10 million downloads. We find a sensitive flow that starts in the watch with source *getLatitude()* from the Location API, and it is sent to the mobile app with the `MessageClient` API using the path "request-network-flights". Then, the mobile app sends out this data through URL using the *HttpURLConnection* object and writes it to the file system using the *java.io.OutputStream* class. In this case, the wearable and the mobile declare the location permission in the Manifest. However, this alone is not enough to comply with the guidelines.[3] In this case, the wear app must send the user to the phone to accept the permission. This case shows that it is possible to bypass the permission system using `Data Layer` APIs.

Permission re-delegation. Venom (*fr.thema.wear.watch.venom*) is a Watch Face customized for a watch user interface. The mobile version of this app uses the *android.database.Cursor* class to store sensitive information such as the call history or unread messages in a database. The app aggregates all information in a `DataItem` object and synchronises it with the wearable app. However, the wearable app does not declare the relevant permissions. Interestingly, the string analysis has been key to uncover the type of information the app is retrieving from the database and to trace it back to API sources that relate to the sensitive information discussed (e.g., missed calls and text messages).

Sensitive Data Exposed Finally, we observe evidence of apps exposing sensitive data through a wide range of sinks, including the Android Broadcast system and Shared Preferences. For instance, Talent (*il.talent.parking*) — which is used for car parking — reads data related to the last parking place and its duration from a database and synchronises it with the watch using a `DataItem`. Then the wearable app writes the data to Shared Preferences. Another example is the app *com.mobispector.bustimes*, which shows bus and tram timetables, and has more than 4 million downloads. The app reads data from an HTTP response, then send it to the wearable through the `MessageClient` API, and finally executes a system Broadcast exposing the content of the HTTP response.

All these cases show how developers leverage the `Data Layer` API to send sensitive information. While it is unclear whether or not these cases intentionally use Google Play services to hinder

[3] https://developer.android.com/training/articles/wear-permissions

the detection of data leakages, we see that `WearFlow` is effective at exposing bad practices that can pose a threat to security and privacy.

4.6 Discussion and limitations

This section outlines the limitations of our work. These may arise from WearFlow's implementation or the dataset we used.

Data Transfer Mechanisms. `WearFlow` inherits the limitations of static analysis, i.e., it is subject to constraints of the underlying flow and string analysis techniques. This means it fails to match data flows with native code, advanced reflection, or dynamic code loading. Still, `WearFlow` can be used together with other frameworks [40, 41, 88–91] that handle these issues to improve the accuracy of the analysis, as we discussed in Section 2.3.3.

`WearFlow` considers obfuscation while performing the analysis of apps. We discussed the trivial and non-trivial obfuscation techniques commonly used in the wild in Section 2.3.3. `WearFlow` type-signature deobfuscator is resilient to all trivial techniques and five non-trivial, but it fails to deobfuscate APKs with class or package renamed and reflection. As mentioned before, `WearFlow` did not found relevant APIs in 20% (651) of the APKs, many of which correspond to APKs with these obfuscation techniques.

A more robust deobfuscation technique should rely on other invariant transformations such as the hierarchical structure of the classes and packages. Many methods of the Data Layer library are stubs, therefore using features of the method body will not improve the accuracy drastically. Additionally, one could obfuscate those features (e.g., using reflection together with string encryption) introducing false negatives.

In our case, we have successfully leveraged system types to disambiguate methods with the same signature. We chose to only use system types as they are less prone to obfuscation than other data types and have helped reducing the number of false positives. One could take into account the threat model, and chose to use a more coarse type of signatures when certain traits (e.g., reflection) appear in the app.

Branching. Another potential source of false negative data flows stems from the backward analysis, in the context extraction. `WearFlow` stops the backtracking when encounters a definition of a channel, but this definition could be part of the branch of a conditional statement. Depending on the scope of the variable, the channel could be defined in another method or even another component. Finally, if the string analysis is unable to calculate the value of a key or path (e.g., when there are multiples values due to branching) we use a wildcard value, i.e.: we match any string. This means that the matching step will overestimate the potential flows between the entry and the exit point.

Dataset. Our dataset is limited to 3,111 APKs and 220 package names after considering different APK versions. There are more than 220 apps available for Wear OS, however, identifying

them is a challenging task. Google Play does not offer an exhaustive list of apps with Wear OS components, nor it is always featured in the description of the app. This restriction limits our ability to query Wear OS apps in Google Play. Furthermore, datasets like Androzoo do not provide information about whether a app has wearable components or not. Thus, we need to download apps as the rate limit allows. Given the low density of these kind of apps in the overall set of Android apps, the amount of apps that can be obtained this way is very limited.

Model Accuracy. The precision of the analysis also depends on the accuracy of the `Data Layer` model. The `Data Layer` model used by `WearFlow` replicates the `Data Layer` model, as described in Google's Wear OS documentation. If the `Data Layer` APIs were to transfer data trough undocumented components of the OS or even through the cloud (e.g., via backups), `WearFlow` would not detect such flows. Also, our model is based on Wear OS versions 1 and 2. Wear OS is under active development. Thus, any new APIs introduced in future versions will need to be modelled.

4.7 Related Work

Mobile-Wear communication can be seen as a kind of inter-app communication where one of the apps is being executed in a wearable device. Several works have focused on app collusion detection [36, 43, 68, 136–138]. These works model ICC methods to identify sensitive data flows between applications running on the same device.

`WearFlow` complements existing works by extending the analysis of these apps into the Mobile-Wear ecosystem, increasing the overall coverage of these solutions to all current app interactions in the Android-Wear OS ecosystem. One may argue that these tools could be extended to cover for Wear OS interactions. As an example, works such as DialDroid [39] uses entry and exit points to match ICC communication between mobile apps. In our case, we consider wearable APIs as sources and sinks, which could be easily replicated in DialDroid. However, These APIs aggregate multiple data into a single API call, and we need to match data types on the sender and receiver side, which would lead to inaccuracies in DialDroid and many other tools [139]. ApkCombiner [140] combines two apps into one allowing to run taint tracking on a single app. This approach does not allow us to reason about individual items aggregated into a single API call.

There has been a recent interest of the community in expanding the scope of data tracking to more platforms outside the Android ecosystem. Blasco *et al.* present a survey that reports the security issues of wearables. Zou *et al.* [15] studied the interaction of mobile apps, IoT devices and clouds on smart homes using a combination of traffic collection and static analysis. They discovered several new vulnerabilities and attacks against smart home platforms. Berkay *et al* proposed a taint tracking system for IoT devices [16]. `WearFlow` could have been implemented following the same approach (analysing WiFi and Bluetooth communications between Android and Wear OS). This would have required us to reverse the different communication protocols and data exchanged in both wireless protocols. Our approach is simpler and does not require additional hardware to execute.

Recently, Dini *et al.* presented a user awareness study on fitness tracker data [141]. They investigate whether collected data can be used to infer routines or other sensitive information. Then they utilise a privacy tool to educate users about privacy risks. In a similar line, Psychoula *et al.* developed a framework to integrate privacy user awareness in wearables and IoT devices development process [142].

4.8 Chapter Summary

In this chapter, we have studied information flows in the wearable ecosystem. Based on the characterisation of the previous chapter, we identified the `Data Layer` API as a channel to transmit sensitive data across mobile and wearable apps. This channel is not available on other platforms, presenting a platform-specific issue RQ2. Current state-of-the-art approaches fail to model the complex interactions offered by the `Data Layer`. This results in either missed information flows or a complete overestimation of sensitive transmissions. To address this limitation, we develop `WearFlow` RQ1.

`WearFlow` is a static analysis framework that systematically detects potential data leaks in the Mobile-Wear Android ecosystem. `WearFlow` augments the capabilities of previous works on taint tracking, expanding the scope of the security analysis from mobile apps to smartwatches. We addressed the challenge of enabling inter-device analysis by modelling Google Play Services, a proprietary library. Our framework can deal with trivial obfuscation and most of the non-trivial obfuscation techniques commonly used in the wild.

We have created WearBench, the first benchmark for analysing inter-device data leakage in Wear OS. Our evaluation shows the effectiveness of `WearFlow` over other approaches. We also analyse real-world apps in Google Play. Our results show that our system scales and can uncover privacy violations on popular apps, including one with over 10 million downloads.

5 Security and Privacy of the Android TV ecosystem

In this chapter, we present a security and privacy analysis of the Android TV ecosystem. We analyse the behaviour of TV apps in terms of sensitive data collection and communication with with other devices using a pipeline of static analysis tools, network traffic collection, and verification via manual analysis.

5.1 Introduction

Currently, Smart TVs are the most popular smart devices after smartphones [2, 13]. There are more than 110 million active Android TV devices, which have seen an 81% increase in viewing time per year since 2020 [143]. Smart TVs offer enhanced capabilities and connectivity in comparison to traditional TVs. However, the additional capabilities also introduce vulnerabilities [2, 13, 14, 26, 27] and potential privacy violations [17, 18, 25]. Users' Personal Identifiable Information (PII), such as unique identifiers, hardware addresses, or viewing habits, can be exposed to TV providers and analytics/advertising services.

Android TV is widely available across TV brands and shows high integration with Android, which facilitates consumer adoption but also access to their applications for analysis. Recent works [28, 29] have focused on traffic analysis and the detection of tracker domains on Roku and Amazon Fire platforms. We are interested in measuring the Android TV ecosystem and understanding the threats to users' security and privacy. We use a mix of static and dynamic analysis for these purposes in this chapter. We proposed `WearFlow` to study static information flows in the previous chapter. The main advantage of `WearFlow` is the precise modelling of mobile-wear communications. As the `Data Layer` is not available in Android TV, we develop a custom pipeline that we describe in section 5.5.

In this context, we want to identify privacy-invasive behaviours in TV apps and identify the stakeholders behind such practices. Additionally, we want to analyse the interaction of TV apps with other devices and their relationship with tracking and advertising services. In particular, this chapter makes the following contributions:

- We provide a detailed study of the Android TV ecosystem. We collected approximately 4.7k TV apps from different markets. The dataset is four times larger than previous evaluations [28, 29], and it contains streaming apps and a myriad of TV-compatible utility apps and games that have proven to be equally invasive regarding sensitive data collection. Additionally, we compare the TV and mobile version of popular streaming apps and present

the main differences between them.

- We develop a static analysis pipeline for analysing TV apps complemented with traffic analysis experiments and manual analysis. We find that most TV apps present potentially harmful and privacy-invasive behaviours and that tracking and advertising libraries are responsible for a significant part of these cases.

- By analysing inter-device communication patterns in TV apps, we found that developers typically rely on old APIs to communicate with other devices and do not implement best practices to secure communication channels when using new APIs. We observe that around 50% TV apps include the code to open ports; this number is significantly greater than previous reports in the mobile ecosystem [128, 129].

- We find malware and apps with invasive behaviour in our dataset. We responsibly disclose our results so security and privacy can be improved in the app ecosystem. We notified Google of applications having abusive practices not included in their privacy policies. We also notified developers using non-secure practices so they can fix their apps and make them more secure.

5.1.1 Security Model and Threats in Android TV

Android TV apps are subject to the same security model as Android mobile apps. We refer the reader to Section 2.2 for an overview of the security model of Android apps. We highlight that manufacturers and developers can create custom permissions to protect their apps' resources or share them with other apps.

Developers of TV apps can use third-party libraries (TPL) to add services or functions to their apps. Such services include analytics, advertising and social networks, among others. Libraries inherit all permissions from the host app, which might give them access to sensitive data. Previous works reported multiple TPL collecting sensitive data such as unique identifiers, geolocation, and user data [19, 22]. More concerning, developers might be unaware of these data collection practices and other threats to users' privacy [144].

Previous studies on Smart TV applications have focused on applications available for other platforms such as Roku and Amazon Fire TV [28, 29][1]. In contrast, we analyse the Android TV ecosystem to provide the community with a better understanding of the security and privacy risks of Android TV and compare the risks of these apps with their mobile counterparts. In particular, we consider the following threats:

- Bad development practices. This includes incomplete or false information in signing certificates and permission misuses, such as inconsistent definitions caused by porting code across platforms. Applications might request more permissions than they need [145, 146],

[1] While Amazon Fire TV is heavily based on Android, their apps cannot make use of all the APIs available on Android TV.

increasing the attack surface on Smart TVs, amplifying the impact of bugs and vulnerabilities, and violating the principle of least privilege [23].

- Sensitive data leakage. Particularly PII collected by third-party services. TV apps also have access to personal data specific to Android TV, for instance, the title and genre of programs defined. This information about users' viewing behaviour can be used for user profiling [13, 29, 147]. A previous study shows that more than 70% of Smart TV users consider the commercialisation of their viewing history unacceptable [30].

- Vulnerable inter-device communications. Such communications pose security and privacy implications for consumers [13,27]. Smart TV users are exposed to eavesdroppers listening to unprotected communications that can be exploited by attackers, as shown in previous work [148].

- Malicious applications. TV apps can present malicious behaviour. This can be due to developers adding malicious code or malware exploiting some vulnerability [54, 149].

5.2 Dataset Collection

Identifying and collecting TV apps at scale is challenging because the official Android market does not provide platform-specific metadata (TV, Mobile, Wear). Therefore, we use an initial seed of well-known TV apps to search for related apps in the Play Store. We collect all identified package names and then download the last version of each app (as of August 2020) from AndroZoo [135]. This repository is a well-known source of Android apps widely used in the literature [33, 150–152] that includes a `VirusTotal` evaluation summary. We inspect the manifest of all downloaded apps to verify they can be run on Android TV, as described in Section 3.3. We also search for TV apps in APKMirror [153] using the keyword *Android TV* and download them with a crawler. In total, we collected 4745 TV apps with unique package names from both repositories.

	Touchscreen disable	TV Activity	TV Libraries	TIF
TV-only	√	√	√	×
TV-enabled	√	√	×	×
TV-streaming	√	√	√	√
Mobile-streaming	×	×	×	×

Table 5.1: Categorisation of apps included in our dataset.

We classify the TV apps into four categories according to the configurations shown in Table 5.1:

- `TV-only`. This group includes apps that can run exclusively on Smart TVs. They require: 1) touchscreen disabled 2) declare a TV activity 3) include libraries that are specific to Android TV (Leanback).

- `TV-streaming`. The primary purpose of these TV apps is to show streaming content.

These apps require the same configuration as the TV-only apps, plus they implement the TV Input Service. We inspect the APK bytecode with Androguard to detect the corresponding API calls within the apps' components.

- `TV-enabled.` These apps can run on Smart TVs or other devices like smartphones. Apps in this category declare a TV activity and the touchscreen disabled.
- `Mobile-streaming` includes mobile apps whose primary purpose is to show streaming content, for instance, the Netflix mobile version. These apps do not require any setting from Table 5.1.

It is worth mentioning that we assign a TV app in the `TV-streaming` category if the app implements the TIF. Although the TIF is the standard way to communicate with a media provider, it is possible that some developers use a custom implementation for this interaction.

Table 5.2 shows a summary of the apps in the dataset. Even though our focus is on supposedly benign TV apps, there are 34 TV APKs that are flagged as malware in `VirusTotal` by more than 5 antivirus engines, of which 60% are considered malicious by more than 10. We use the malware threshold similar to other works [150, 151]. We expand the discussion of malware in Section 5.7.

TV-only	TV-enabled	TV-streaming	Mobile-streaming
831	3911	100	90

Table 5.2: Dataset classification. Note that one TV app can be in the TV-only and TV-Streaming group at the same time.

5.3 Ecosystem Overview

This section provides an overview of the ecosystem of Android TV, which includes developers, permissions, and third-party libraries, including tracking and advertising services. Our preliminary analysis is based on features extracted via static analysis, supported by manual verification in some cases. This analysis is intended to bring to light bad development practices that can affect users' security and privacy. We also aim to identify analytic services and use the list of trackers as input for the information flow experiments.

5.3.1 Developers

We start the ecosystem analysis by examining the developers behind the app's signing certificates. This information allows us to attribute apps to the same developer and identify the ones that incur heavily on privacy-invasive behaviour. While self-signed certificates cannot be trusted as an identity source, they can be used to identify apps that have been developed by the same organisation or developer.

First, we use Androguard to extract the SHA256 fingerprint from the certificates to identify unique developers. Then, we use the field `Issuer` to attribute one certificate to its organisation.

In total, we identify 3623 unique fingerprints. 75% of the developers own only one app in our dataset. In the remaining group, 50% publish two apps and only 12% more than five apps. This trend shows that most developers publish one or two TV apps. Similar results were obtained by other studies [154, 155].

Considering the `TV-streaming` category, there are only 4 developers with multiple apps. Likewise, developers from the `TV-only` group tend to publish a single app, and developers with multiple apps correspond almost exclusively to the game category. We further investigate if there are developers in the `Mobile-Streaming` and `TV-only` and found that the groups are largely disjoint, with only 3 developers in both groups.

Interestingly, we found 85 APKs without information about the signing organisation, including empty fields or completed with irrelevant strings. 1234 apps use the same information as the Google apps excluding the signature. We check this list against the apps published by Google in its Play Store profile and determine that Google indeed develops only 30 of these apps. Previous works [155, 156] reported similar issues in the mobile ecosystem. While it is unclear if developers try to mislead analysts, the Play Store, avoid accountability, or it is just a bad development practice, the results indicate that this behaviour is prevalent in the Android TV platform.

5.3.1.1 Malicious Apps in Android TV

We detected 34 APKs in our dataset that are flagged as malware by more than 10 engines in `VirusTotal` (VT). 6 of these APKs are from the `TV_only` category, of which one was removed from the Play Store, three are paid apps, and two are VPNs. Notably, the APKs with higher VT scores (25-34) are games. These APKs include payloads capable of remote execution, capture inputs, sensitive data collection, fraudulent ads, and silently install packages according to the `VirusTotal` evaluation.

We attribute the APKs with malware to 22 developers using their certificates. Based on this, we were able to identify an additional set of 44 APKs belonging to these developers (10 `TV_only`). Most of the added APKs have a low VT score. However, there are some noteworthy cases. For instance, we found the `TV China` app that offers IPTV services under two different package names (`com.live.tv.home` and `com.cntv.movies`). The VT scores are 7 and 9 respectively, and their evaluation is almost identical, with 7 months of difference between the evaluations. Both apps were removed from the Play Store by the time of submitting this thesis.

These examples give evidence of malicious behaviours in Android TV. There have been a few examples of malware developed for Android TV in the past [115]. Recently, Zhang *et al.* show that it is possible to attack Smart TV devices by remote control imitations [157]. It remains to be seen whether custom malware for Android TV appears on a large scale and if they target or exploit platform-specific resources. However, there is no doubt that TV users are susceptible to malware threats.

5.4 Permissions

We leverage Androguard to extract and analyse permissions from the TV apps. We identify a total of 225 AOSP permissions and 2600 custom permissions. Although the number of permissions per app varies greatly, Table 5.3 shows the average and maximum permission request per type. Next, we offer a glimpse of permission requests by the TV apps in our dataset.

	AOSP		Custom	Total
	Normal	Dangerous	-	-
avg	6.05	2.1	4.7	12.9
max	91	20	123	234

Table 5.3: Permissions summary

AOSP Permissions. Figure 5.1 shows the top 10 dangerous and normal permissions requested by TV apps. As one might expect, permissions related to network connectivity are at the top of the list. TV apps request 6 normal and 2 dangerous permissions on average. Although we focus on the use of dangerous permission, we discuss in section 5.7 some aspects of normal permission.

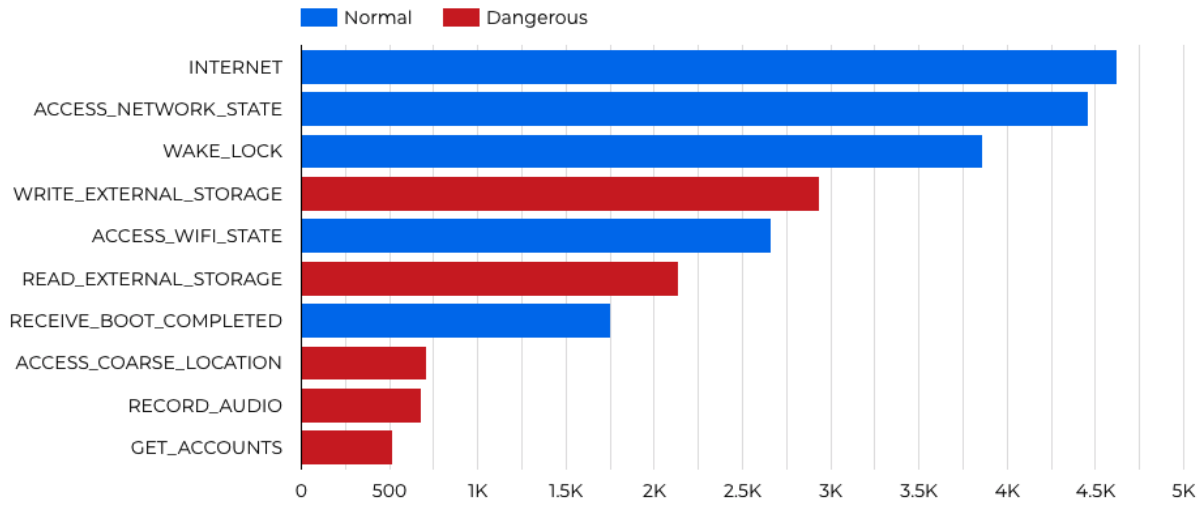


Figure 5.1: Top 10 normal and dangerous permissions

Dangerous Permission. These permissions protect the most sensitive system resources. The most requested dangerous permissions are related to external storage, location, record, audio, and account information (see Figure 5.1). We note that multiple TV apps request permissions not available in Android TV, such as telephony-services, and sensors. This peculiarity suggests that developers are porting mobile apps to TV without considering security aspects, leading to inconsistent apps. Manual validation confirmed that TV apps request permission even if the hardware is not available, e.g., microphone.

Custom Permissions. We identify 2600 unique custom permissions requested across all APKs. 1809 permissions are declared by apps present in our dataset, while external apps declare the remaining 791 permissions. Considering `TV-enabled` apps, around 35% declare and 90% request at least one custom permission. 87% are signature-level permissions, and the remaining part is

divided between mixed, normal, and dangerous in this order. Previous studies give the number of apps using custom permissions between 25% and 65% [158,159]. We attribute this difference to the target apps included in this study (Android TV). Our work shows that Android TV apps normally require resources from other apps to work. Interestingly, most apps do not complete the description field with meaningful information. The description field should describe the behaviour in case the permission is granted or denied [160]. Over 94% of the apps leave this field empty, including four apps declaring dangerous custom permissions.

5.4.1 Third-Party Libraries

In this section, we analyse the presence of third-party libraries (TPL) in our dataset. Identifying TPL allows us to distinguish between data exposure to first/third parties, where the latter represents a more significant risk to user's privacy [24,29]. Additionally, it allows us to characterise tracking and advertising services in the Android TV ecosystem.

We use LibScout and LibRadar similarly to the preliminary evaluation in Chapter 3. Additionally, we implemented a lightweight library detector using Androguard. Our approach consists of several steps: 1) Disassemble each APK, extract all package name prefixes, and filter out irrelevant packages, e.g., well-known Android packages and app components. 2) Cluster packages by prefixes and keep the packages that appear in at least 10 apps. 3) Classify packages using the categories from previous works [19,22] and Privacy Grade [161]. With this, we aim to detect libraries not present in the repository of the above-mentioned tools. Note that this is a best-effort approach, and it is not resilient to obfuscated APKs. We discuss the limitations of our approach in Section 5.7.3.

Table 5.4 summarises the libraries identified in our dataset. We detected 843 TPL in total, of which 64 are not present in other well-known repositories [19, 22]. We detected Social Media libraries in 88% of the apps, with multiple Facebook libraries occupying the top 5. Similarly, we found that 75% of the apps contain analytics libraries and 77% contain advertising libraries. 66% of the apps include advertising, analytics, and social media libraries together. Our analysis detected cloud libraries in 223 APKs, most of them owned by Amazon.

Category	# Libs	# Apps	Examples
Advertising	47	3637(77%)	Ogury, Appodeal
Analytic	16	3595(76%)	Firebase-analytics
Cloud	23	210(4%)	Amazon-Kinesis-Streams
Social Media	10	4195(88%)	Facebook, Twitter, VK
Utilities	747	4591(97%)	OkHttp, Stetho
Total	843	4745	

Table 5.4: Third-party library summary

Some streaming libraries not detected by previous research are Penthera and Vizbee. The former is a library that facilitates the delivery of video content for mobile apps, and Vizbee is a library that allows mobile apps to stream video to a TV via a native app. Vizbee's website mentions that they collect installed applications and constantly scan networks. They claim the possession

of a graph of available devices across millions of homes, in which they can distinguish between private homes and public places [162]. Another library for mobile and TV apps is Nielsen SDK, which enables measuring live and on-demand TV viewing behaviour. We found this library in many TV apps such as Hulu and Player. We detected data flows reading sensitive data in all cases (more details in Section 5.5).

We focus particularly on analytics and advertising libraries given recent ad spending spikes in Android due to changes in iOS privacy policies [163]. In this context, Google Play Services ads (2502 APKs) and Firebase analytics [2] (2407 APKs) are the most popular libraries. Other ad libraries found include Vungle, Tapjoy, and Jirbo. Smaato and FreeWheel are platforms specialised in video advertising and connected TV services.

In total, we detected 70 tracking and advertising libraries, many of which were linked to data violation policies in the past [164, 165], some examples are Appodeal (84 APKs) and Umeng (27 APKs). Our static analysis shows that a large number of data flows correspond to these libraries and we were able to confirm these findings with network traffic analysis experiments.

We also detected atypical libraries such as Javassist (11 APKs), that could affect the accuracy of static analysis. This library can modify Java bytecode and obfuscate Android apps, which is the case in some TV apps that we manually analysed, e.g., `com.funkidslive.action`. Similarly, previous studies found that the game engine Unity reads and uploads hardware addresses to a remote server using covert channels [47] and persistently requests information about installed apps [144]. Our static analysis results indicate that this library might be collecting similar information in 278 APKs.

5.5 Behavioral Analysis

Our preliminary analysis provides intuitive information about specific app features such as permissions and their libraries. However, this information is not enough to determine if TV apps are capable of disseminating sensitive information or using vulnerable communication channels. In this section, we use static analysis tools to analyse these threats, and we verify some of our results with dynamic analysis and network captures.

5.5.1 Intra-device Analysis

Static Analysis. We use taint tracking to detect potential sensitive data leaks. This analysis gives information about the propagation of data from sensitive sources to sensitive sinks. We use Flowdroid [32] because it provides a good balance between accuracy and performance on real-world apps [39, 79]. Note that for TV apps, we cannot use `WearFlow` as the `Data Layer` API is not available on Android TV. Instead, we customised Flowdroid by adding sources and sinks that are specific to Android TV. The sources include media metadata, identifiers, and other methods added in the latest Android releases. Additionally, we added callbacks from the

[2] Renamed Google analytics 4

Leanback library and other media-related libraries to enable the creation of a more accurate control flow graph.

Dynamic Analysis. The traffic analysis experiment executes TV and mobile apps on real devices and then analyses their network traces, searching for identifiers and user-sensitive data. We rely on *Charles* [166], an HTTP proxy, to intercept and decrypt network traffic. Figure 5.2 illustrates an overview of our methodology. We first instrument the APKs using the *mitm-proxy* script [167] to add Charles's custom certificate and try to remove certificate pinning checks in the bytecode. Additionally, we search the main TV Activity in the Manifest and add statements to log static identifiers at the beginning of the `onCreate` method. We later search these identifiers in the network traces. The instrumentation is only necessary for TV apps because there is no way to install custom certificates on Android TV. Then we install and run the APK on the TV device using the Android Debug Bridge tool (ADB) that comes with the Android SDK. Previous work [29] showed that apps contact more third-party domains after login. So, we created testing credentials to log in manually on each app if the application does not require a paid subscription. Then we explore the apps for around 15 minutes trying to stream content and trigger ads. Finally, the network traces are stored for further processing.

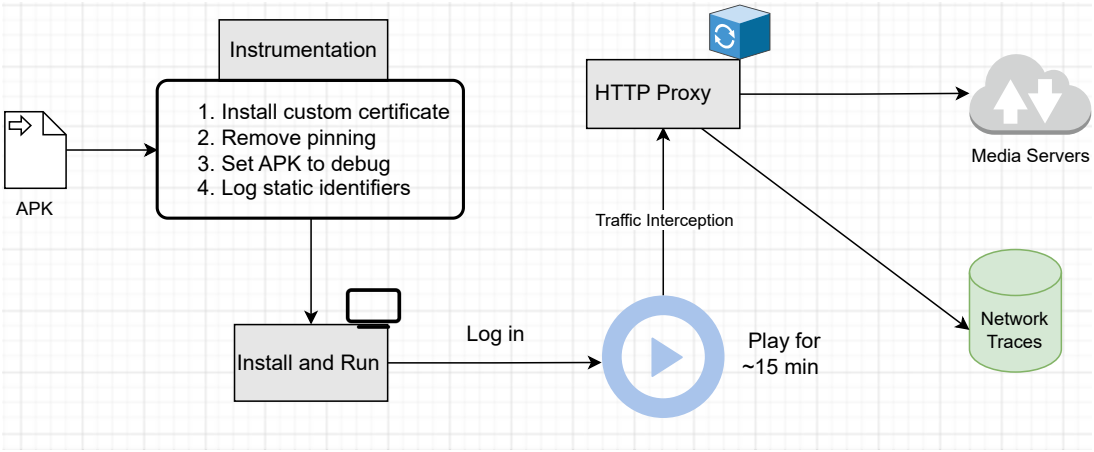


Figure 5.2: Traffic analysis flow

5.5.2 Experimental Results

We run the static analyses on the entire dataset of 4.7K APKs plus the 65 popular streaming apps we used in Section 3.3. For the dynamic analysis, we only use the latter. We also use these APKs to compare the TV/mobile version of apps (Section 5.7). Note that we only run the traffic analysis on the `Streaming-popular` group as manual exploration does not scale, mainly because we decided to log in on each app during the testing and the exploration timeout. The list of tested apps can be found in Table 3.4.

5.5.2.1 Static Analysis

We run Flowdroid with a timeout of 8 minutes per analysis. The experiment was conducted on a server with 46 cores Intel Xeon CPU E5-2697 v3 @ 2.60GHz, and 92 GB of memory. The analysis failed for a small number APKs. 29 timed out after doubling the original time limit. In

our experience, analysis that surpasses this timeout runs for hours. Another 18 failed for other reasons such as errors while parsing APK resources or unexpected bytecode.

The analysis found at least one sensitive data flow in 78% of the files. The number of sensitive data flows varies greatly: for instance, 313 APKs contain only one result. In contrast, 470 APKs contain more than 50 sensitive flows, and we could observe 78 APKs having more than 100 on the extreme side. The percentage of sensitive data flows per app category is as follows: TV-only (81%), TV-enabled (75%), TV-streaming (83%), and Mobile-streaming (92%).

Figure 5.3 shows data flows for a selected number of sources. We limit the output due to space constraints and to help with visualisation (full results are shown in tables 5.5 for sources and 5.6 for sinks). The diagram shows that identifiers, location, and package manager information are the most popular sources. The categories Tracking_ID and Hardware_ID include values such as device identifiers, MAC addresses, and other values that can be used as unique identifiers or for fingerprinting. Geolocation and Wireless_ID offer more information for tracking and profiling. The User_Info category includes account information such as email and private user data. Last, the Package_Manager provide information about installed applications and modules, hardware capabilities, shared libraries, and other system information. We chose these categories as they represent a direct threat to users' privacy and because we want to compare the static analysis results with the traffic analysis experiments.

Additionally, we found data flows in 1256 APKs logging sensitive data, of which 80% correspond to TPL. For instance, we detected data flows corresponding to the Kochava SDK logging UUIDs, SSID, and geolocation in popular apps like Sling and NBC Sports. Even though other applications cannot read this information in a typical situation (the Logger is protected by permissions), logging sensitive data is discouraged by Google because of risk exposure and performance [168].

Tables 5.5 and 5.6 show that a large percentage of sensitive flows happen in TPL, including categories such as Tracking_ID (52%), Hardware_ID (45%), and Wireless_ID (48%). Note that many flows reach non-internet sinks (Table 5.6 and Figure 5.3). These channels do not necessarily indicate exfiltrations, but they could expose data to shared resources (e.g., system broadcast) that is leaked by a second data flow. For instance, we detected 1398 extra data flows using `SharedPreferences` and File API methods as connectors. Although these are overestimations of potential flows, they still show the need to analyse such cases. Understanding such data flows requires more complex analysis. Reardon *et al.* provide a large-scale evaluation of side and covert channels in Android [47]. Now, we focus our discussion on selected categories that are common targets of abusive and malicious developers and the prevalence of trackers.

Identifiers. It is well known that tracking and profiling are pervasive in the Android ecosystem [17, 18, 23, 155]. We investigate what identifiers are being used by TV apps by checking the sources. The most used identifier is a globally unique ID (GUID) generated with the java.util.UUID package (3031 APKs). The most common sinks for this source are Logs (48%), Shared_Preferences (33%), File_Write (8%), and Network_Traffic (5%). The sources are collected by the app's component only in 10% of the cases, while advertisement libraries are responsible

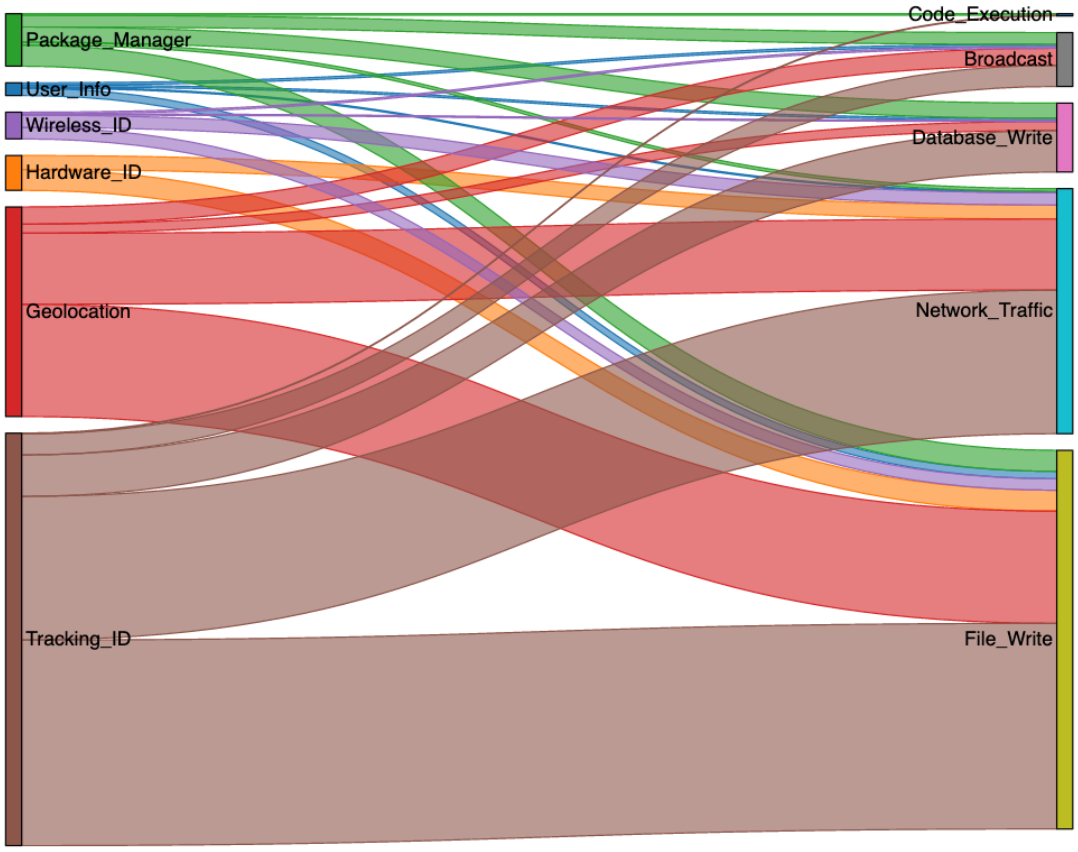


Figure 5.3: Sankey diagram of sensitive data flows

for 72% of these data flows.

We highlight that resettable identifiers are largely absent from our results. In contrast, we found 285 APKs with data flows reading MAC-Address and SSID identifiers, of which around half of these cases correspond to TPL, despite the Google recommendation of not using hardware identifiers for tracking purposes [169]. Most of the APKs (88%) that use static identifier sources share them with at least one third-party library. For instance, some destinations are the Facebook SDK (58% of these apps), Tapjoy 11% and Presage 4%, which are well-known trackers.

User Private data. The most requested sources from the category `User_Info` are the list of accounts, followed by display-name and email. The most popular sink categories are Logs (66%), File_Write and SharedPreferences (11%). There are also a few instances of Network_Traffic and system Broadcast. Around half of the cases correspond to information flows in third-party libraries. 725 APKs (18%) contain information flows from the `Package_Manager`, including the list of installed packages (169 APKs). 83% of these calls are made from TPL, of which AppsFlyer (47%) and Bytedance (11%) are the most noticeable.

Although getting the precise geolocation is not always possible for TV apps due to hardware

Category	Method example	count#	apps#	Comp%	Lib %	Indet %
Tracking_ID	UUID: randomUUID()	34395	3031	10	52	38
User_Info	GoogleSignInAccount: getAccount()	294	154	32	42	23
File_Read	File: getCanonicalFile()	1256	961	15	75	10
Hardware_ID	WifiInfo: getMacAddress()	1018	277	28	46	26
Wireless_ID	WifiInfo: getSSID()	399	113	40	49	11
Database_Read	Cursor: getString(int)	43551	4392	35	49	16
Package_Manager	PackageManager: getInstalledPackages()	1677	725	25	56	19
Media_Info	AudioRecord: read(byte[],int,int)	112	31	71	23	6
Network_Connection	URLConnection: getInputStream()	13015	3156	15	48	37
Location	Locale: getCountry()	5279	1612	35	48	17

Table 5.5: Sources of sensitive flows. Comp = Component, Lib = Library, Indet = Indetermined. The Method example column shows a truncated signature: Class: method-name(params)

Category	Example methods	count#	apps#	Comp%	Lib %	Indet%
Database_Write	Cursor: update(params)	5198	689	57	30	13
Log	Log:int d(params)	39449	2918	21	47	32
Shared_Preferences	SharedPreferences: putString(params)	15210	2459	21	35	44
Bundle	Bundle:void putParcelable(params)	4041	1700	23	70	7
File_Write	OutputStream:void write(byte[])	23116	3218	15	69	16
Code_Execution	Runtime: exec(java.lang.String)	19	6	17	33	50
Network_Traffic	HttpURLConnection: getOutputStream()	6359	1453	15	73	12
Media_Out	MediaRecorder: start()	8	4	100	0	0
OS_Messages	Handler:boolean sendMessage(android.os.Message)	3620	768	32	35	33
Broadcast	Context:void sendBroadcast(android.content.Intent)	1509	326	58	25	17

Table 5.6: Sinks of sensitive flows. Comp = Component, Lib = Library, Indet = Indeterminate. The Method example column shows a truncated signature: Class: method-name(params)

limitations, we detected 5279 sensitive flows in 1028 APKs (25%) corresponding to the category `Location`. The most popular information requested is country, timezone, and coarse location, usually employed to customise content. Around 35% of the APKs write location data to the Log or store it in Shared_Preferences.

We detected 31 APKs potentially exposing media information such as channel identifiers and audio recordings. From this group, 12% relates to TV programs IDs, 38% other media metadata, and 41% audio recordings from hardware input. 70% of the cases are in the context of app components, but we also detected data flows that end in libraries such as Facebook and Flurry. After searching for potential data leaks in static flows, we analyse the network traces produced by our dynamic analysis experiment.

5.5.2.2 Network Traffic Analysis

We captured traffic of 21 TV apps and 22 mobile apps out of 30 `Popular-Streaming` apps. In the unsuccessful cases, we failed to collect traffic because the instrumentation broke the app, the certificate pinning modification failed, or geolocation restrictions. Table 5.7 summarises the

number of applications leaking sensitive data to first-party domains, trackers, and CDNs. We use a heuristic based on the Manifest file to infer if the traffic is sent to a first-party domain. In particular, we use sub-strings from the package-name and the app-name fields to search in the destination domain of network traces. We manually validate this classification and adjust it case by case. For trackers and CDNs, we rely on data from previous works [17, 18, 29] and manual analysis.

category	example	1st	Trackers	CDN
Hardware	build fingerprint	11	15	10
Location	latitude, longitude	8	3	3
Wireless	SSID, provider	3	0	0
Static Ids	android ID	5	5	0
TV metadata	Program title	11	5	3

Table 5.7: Traffic analysis. Number of TV apps sending sensitive data to 1st party domains, trackers, and CDNs

Figure 5.4 shows the top domain names collecting sensitive data in Android TV. Although some domains correspond to CDNs, most domains correspond to well-known trackers such as Google's `doubleclick`, and `ScorecardResearch` from Comscore, which is a third-party tracker for measurement of cross-platform audiences, including digital, linear TV, over-the-top (OTT) and theatrical viewership. Next, we analyse the information collected by different trackers and we present a comparison between mobile and TV app traffic in section 5.7.

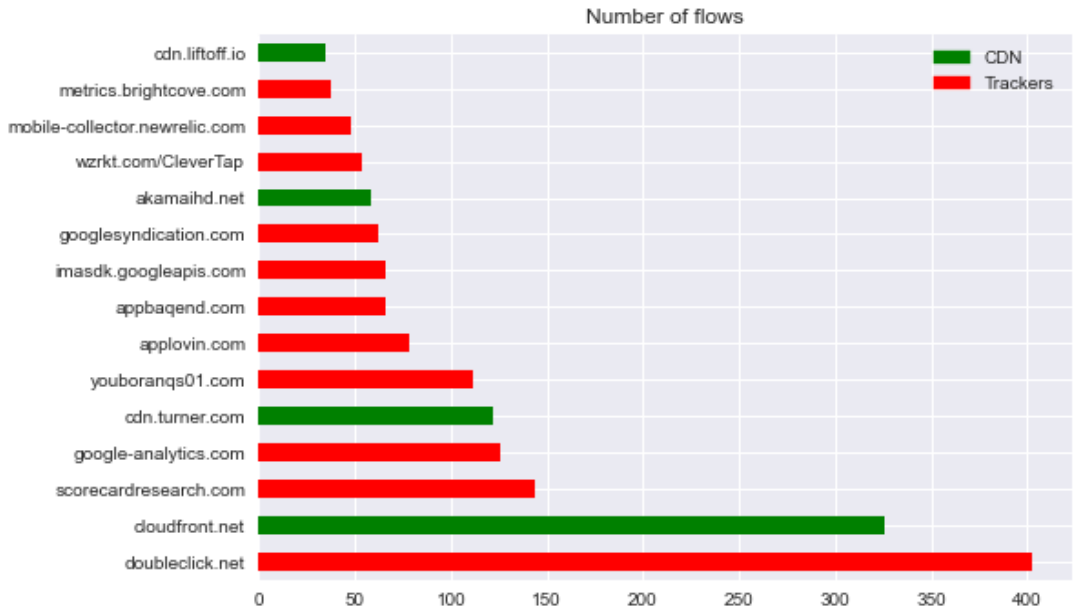


Figure 5.4: Top domain names collecting sensitive data.

Identifiers. We searched static identifiers in the network traces and found that 42% of the explored TV apps (9/21) are using static identifiers. In all but one case, identifiers are sent to third-party tracking domains such as `facebook.com`, `doubleclick.net`, and `yandex.net`. In particular, we detected two groups using the same value as advertising ID (`CBNTV`, `RedBull`, `Hopper` and `AccuWeather` as our first group, and `Radio.UK`, `YuppTV`, `CNN HTB` as the second one).

The latter collects Android ID and appends the advertising ID. In this way, the app can keep tracking even if the user resets the advertising ID. Another example is the `Lifetime: TV Shows & Movies` app. We observed traffic from this app collecting Package_Manager information and the Android ID. These examples clearly show that TV app developers are using static values as identifiers and sharing them with trackers (against Google's recommendation). These values can be used to track user behaviour across apps.

Private data and Viewing history. We observed 10 apps collecting geolocation data. For instance, the live streaming app `HTB` shares the location with `doubleclick.net` and `yandex.ru` domains. We discover that many apps, such as `CNN for Android TV`, use a third-party service (`ip-api.com`) to collect geolocation data based on the IP address. Although this is a legitimate service, it can be used as a side-channel to circumvent the Android permission model and user consent, e.g., apps do not declare the location permission but still collect the user's location.

We detected media metadata in the network traces of 13 TV apps (65%). It is expected to find this type of information (e.g. program title) in traffic to the app's domain and CDNs that provide multimedia content. However, several TV apps e.g., TedTalk, WWE, YuppTV, AcornTV and CNN share this information with well-known trackers such as `doubleclick`, `Conviva`, and `CleverTap`. We observed the full title leaked in many cases, but we also found modified titles, which makes automated analysis more difficult. These results confirm our findings from the static analysis experiments. TV apps collect sensitive data and share it with third-party services. The sensitive data include static identifiers and private user data, including viewing history.

Static VS Dynamic Analysis. The results from both analyses complement each other. Considering the `Streaming-Popular` category (65 APKs), the static analysis detected sensitive flows in 46 APKs, while the dynamic analysis captured network traces in 43. The numbers for each source category vary greatly. For instance, the dynamic analysis was more effective at detecting TV metadata flows (13 vs 2 APKs). In contrast, the static analysis found more flows (14 vs 10 APKs) for the location category. Overall, we use the static and dynamic analysis result to flag apps that we consider inappropriate. For instance, static analysis results show evidence of potential data leaks involving multiple tracking libraries for the app `com.client.sov.adventure`. After manual inspection, we also note that this app collects network information. We confirmed that this app leaks device identifiers and geolocation to third-party domains with dynamic analysis, and it shows excessive traffic from 14 tracking domains. None of these behaviours are described in their privacy policy. We reported this and other apps in the **Report Inappropriate Apps** section of the Play Store. Now, we look at the results of static and dynamic analyses and compare them across app categories.

5.5.2.3 Difference Between Apps Categories

We detected 2910 sensitive data flows in 90 `Mobile-Streaming` apps and 2190 flows in 100 `TV-streaming` apps using static analysis. Both groups share 75% of the source methods. Information collected only by `Mobile-streaming` apps includes the list of installed packages and

Telephony-Service methods. Telephony-Services classes are not available on Smart TVs. However, the absence of installed package sources suggests that streaming TV apps are not accessing this information. In contrast, we found `TV-enabled` apps exposing this information. Unique sources in `TV-Streaming` apps include Time-Zones and TV metadata from the Leanback library, which is exclusively for Android TV. Time-zones are likely used to customise streaming content.

We note similarities between the apps from the `Mobile-Streaming` and `TV-enabled` categories. However, the TV version of the popular apps present notable differences. First, we detected a few TV apps with sensitive flows. 14 out of 25 TV apps (56%) produced sensitive flows, while the other two groups are close to 90%. Second, TV apps contain considerably less sensitive data flows per APK in all but one case. For instance, the Netflix analysis results in 10x more sensitive flows in the mobile version and the Amazon Video in 3x.

We also looked at the difference between the network traces from the dynamic analysis. We found that many apps contact different domains for the same service (e.g., media content and tracking) on different platforms. For instance, `Rakuten` mobile uses `Akamai` as the streaming provider, while the TV version uses the `CenturyLink` CDN. We found differences in the traffic to third-party domains when comparing mobile and TV apps. For instance, `Bitmovin.com` and `ooyala.com` are live streams advertising domains contacted only by TV apps. In contrast, `yahoo.ads`, and `platform.twitter` are domains contacted only in the mobile versions. Mobile apps like `YuppTV` and `WWE` contact social networks domains such as `facebook.com` and `twitter.com` only in the mobile version. A rare example is the tracker domain `scorecardresearch.com`, which is used by the CNN app in both versions. However, we observed traffic to this tracker only in the mobile version of other streaming apps. Finally, Table 5.8 shows a summary of the domains contacted by both versions and the difference in the number of flows and apps. Overall, information flows are different in TV and mobile apps in terms of volume, type of data collected and third-party services.

domain	Description	#TV flows	#TV apps	#Mobile flows	#Mobile apps
akamai	CDN/streaming	573	6	537	7
doubleclick	ads/tracking	44	4	159	9
google-analytics	tracking	51	3	98	6
facebook	ads/tracking	9	1	144	7
scordcardresearch	tracking	36	1	38	4
appsflyer	ads/tracking	5	1	41	2

Table 5.8: Top domains contacted by TV and mobile streaming apps

5.6 Inter-device Analysis

In this section, we evaluate inter-device communication capabilities of TV apps. We focus our analysis on Sockets that can open remote or local connections and high-level APIs for nearby communications. First, we manually collected all relevant methods to send/receive messages using the following APIs: Socket, NearbyConnection, NearbyMessage, and WifiDirect. Then, we search cross-references of such methods using Androguard and classify the context where the

methods are called. The classification is based on the TPL analysis described in Section 5.4.1. Note that our goal is to understand the potential communication capabilities of TV apps and not to measure these connections.

Table 5.9 shows the summary of all communication methods found in the dataset. The preferred communication method is sockets by a large margin. While we found a few cases of the NearbyConnection and NearbyMessage API, we did not find any occurrence of the WifiDirect API.

API	methods	Total APKs	Comp #	Lib #	Both #	Ind #	Total calls
Sockets	send_data	2644	362	2408	249	574	8378
	receive_data	2451	176	2246	113	472	4384
NearbyConnection	send_message	5	5	0	0	0	8
	receive_message	8	8	0	0	0	32
	discovery/advertise	5	5	0	0	0	10
NearbyMessage	publish	3	3	0	0	0	3
	subscribe	3	3	0	0	0	3

Table 5.9: Summary of communication APIs. Comp = Component, Lib = Library, Indet = Indeterminate, Both = Lib and Comp.

.

Socket Communication. Overall, we detected socket API calls in 2646 APKs (56%). Around 90% of the APKs contain socket methods in third-party libraries, while the remaining 10% is split between components and undetermined contexts. Specifically, we detected 382 APKs calling socket methods within app components and a similar number for undetermined contexts. This result indicates that most of the calls to socket methods are made by third-party libraries. In fact, around 87% of the APKs with references to sockets APIs contain references only in libraries.

Regarding app types, the `Mobile-streaming` and `TV-streaming` categories show a high occurrence of socket calls with 92 and 95 % respectively. The number is lower for `TV-only` apps (70%) and decreases even more for `TV-enable` (51%). Previous research [128,129] reported the number of mobile apps with open ports to be between 6 and 15% using static analysis. In contrast, we found that around half of TV apps contain socket communications.

Table 5.10 shows the top third-party libraries that use socket APIs. We were able to identify the libraries corresponding to 90% of the socket calls using the list of third-party libraries previously collected and manual analysis. The library with the most matches is Okio, a library that facilitates file access and HTTP requests.

Figure 5.5 shows an overview of socket usage classified by library category. One could argue that developers are aware of sockets for some categories like communication and security (such as in the case of Okio). However, it is unclear how informed the developers are about these practices considering categories such as advertising and tracking.

Nearby API. We found only 12 APKs using the `NearbyConnection` and `NearbyMessages`

Library	Description	# apps
Okio	I/O, networking	1540
Mintegral	Ads-Analytics	233
Apache	Utilities	227
Facebook	Ads-Analytics	218
Yandex	Ads-Analytics	178
Fyber	Ads-Analytics	169

Table 5.10: Top libraries using sockets

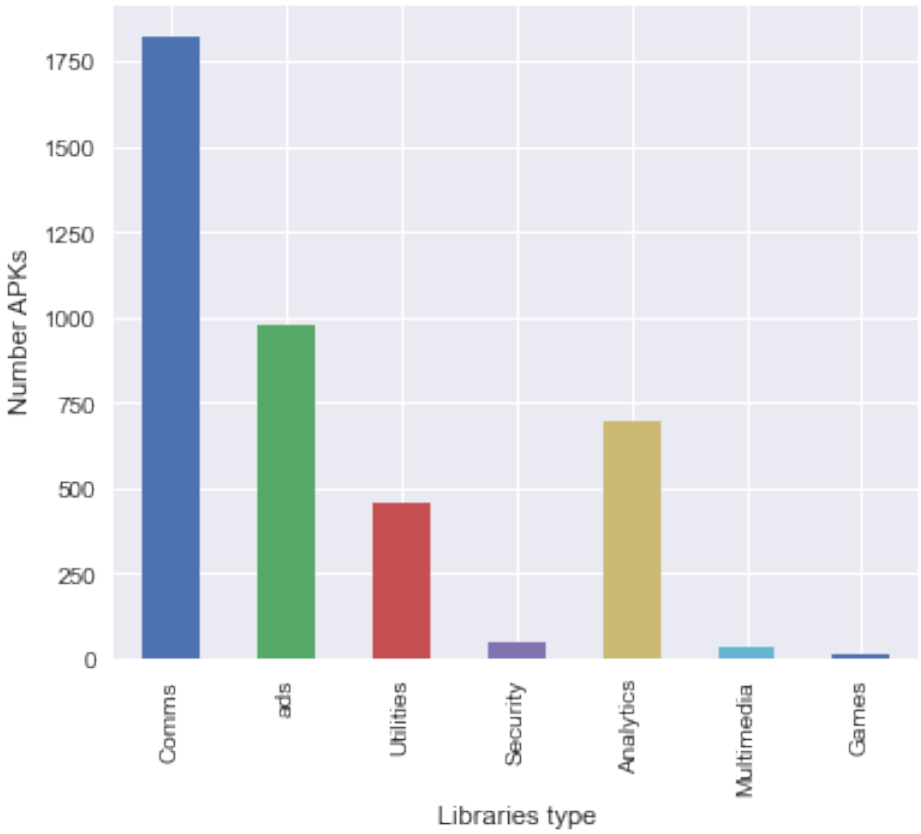


Figure 5.5: Libraries using sockets grouped by category

APIs. Notably, the difference with sockets is that all method calls are made within the app's components. Google guidelines state that connections established without authentication are insecure [130], so we also check whether the apps are implementing authentication by searching calls to the methods that read the authentication token in the callback `onConnectionInitiated`. Unfortunately, none of the APKs that use the `NearbyConnection` API implement authentication. Although this step is optional, a connection without authentication opens an insecure channel that could be exploited [54]. We contacted all developers to understand why they are not implementing the authentication with this API. Unfortunately, we only got one answer which gives little insight.

The Nearby platform APIs are a good alternative to implement two-way communication without requiring an Internet connection, but their adoption is very low in our dataset. Meftah *et al.*

already reported this trend in the Play Store [170]. They argue that constant crashes, battery issues, and testing difficulties contribute to the lack of adoption, which is exacerbated by bad user reviews. Overall, TV apps developers still rely on plain sockets to communicate with other devices.

Old VS New APIs. Many apps we analysed include inter-device communication features. Our work shows that developers prefer to implement sockets rather than the new APIs from the Nearby platform and WiFi Direct. These libraries have been developed since 2014 and have built-in encryption. In contrast, implementing secure communication over a socket is normally more complex and prone to bugs. The fact that developers are not taking up efforts made by the manufacturer to develop more easy-to-use and secure libraries shows flaws in the approach. What is worse, we showed that the few developers that use the newer libraries do not use them properly. We have notified the app developers of this issue. Last, we detected many socket calls in communication libraries, which suggests that the intention is well-known, but there are also instances of tracking libraries, including socket APIs. Although TV apps use sockets predominantly in TPL, the number of socket calls in the app's components vastly exceeds the use of newer APIs.

5.7 Discussions

In this section, we reflect on our findings, how they portray the Android TV ecosystem, and how they are related to the mobile ecosystem.

5.7.1 TV Apps VS Mobile Apps

Our results show that mobile apps request on average 20% more normal and dangerous AOSP permissions than TV apps. This is in line with the additional resources that are unavailable on Android TV. In fact, we found some permissions that are unique to each platform. Permissions related to sensors, orientation, and user dictionaries are found only in mobile apps (and heavily requested). In contrast, TV apps request permissions for capturing TV inputs, content rating systems and HDMI settings.

Since Android TV and Android mobile are highly compatible, many TPLs are used on both platforms. Nevertheless, some libraries are unique to Android TV, generally for codec management and TV-related hardware. Interestingly, it is common to find apps that use different libraries for the same purpose across platforms. From the mobile perspective, we found libraries that facilitate the interaction with Smart TVs showing abusive data collection practices. Regarding sensitive data, the most noticeable difference is that mobile apps rely on precise geolocation. In contrast, TV apps request coarse location and with less frequency.

We observe that TV apps from different categories show similar behaviours, but mobile and streaming TV apps show a higher occurrence of socket APIs than other categories. TV apps differ from mobile apps in that TV apps are updated less frequently. Most of the `Mobile_streaming` apps (94%) were updated in 2022, while for `TV_only`, this number was 75%. Some TV apps

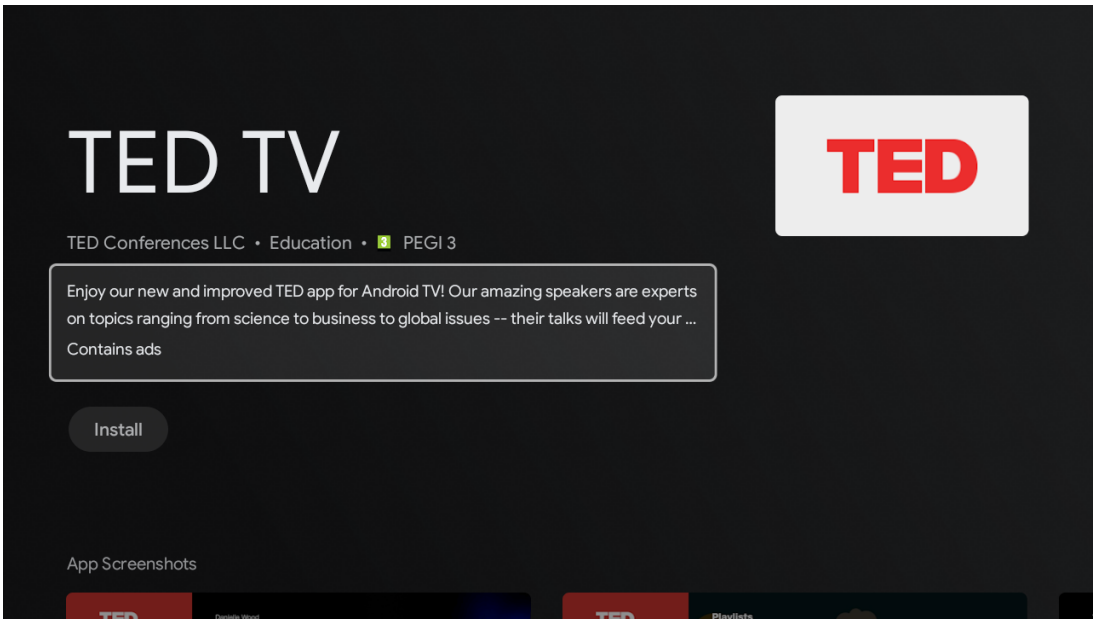


Figure 5.6: Play Store UI

have not received updates for years and are still available in the Play Store. This was not the case for any mobile app in our dataset.

A significant factor limiting user control over privacy threats is that the Play Store UI in Android TV does not provide the same information as the mobile version. Figure 5.6 illustrates this scenario. There is no way for the user to check permissions, read reviews, or the privacy policy of the app. While users can grant or revoke dangerous permissions, they do not have control over normal permissions. In view of the evidence of malicious behaviour and permission misuse, we argue that the Android TV app store should include additional information on their app pages so users can take more informed decisions (at least to the same level as mobile users).

In Section 3.3, we registered a significant degradation in the quality of the TV version of mobile apps based on users' ratings, reviews, and the last update from developers. The permissions analysis from this chapter shows that many TV apps include permissions that are not required by the application or compatible with the platform. This situation arises from porting mobile apps to Android TV without considering components that are not TV-compatible. As these components remain part of the application, users who interact with them get errors. Better developer guidelines and more testing could amend this situation by removing components that are not required at running time and are prone to produce bugs.

Our TPL analysis detected 40% more libraries in mobile apps and more matches per library. For instance, the number of mobile apps with the Facebook SDK doubles that of TV apps. Notably, social media libraries in TV apps in almost non-existent. This might be associated with the UI limitation in Android TV that discourages users from log in to their social media accounts. Even though not all social media apps have their TV version, their SDK can still be used to collect data. While many trackers are common to both platforms, we have found multiple third-party utility libraries for streaming services. Many of these SDKs were not detected by LibRadar

and LibScout because they lack library profiles for such libraries. This gap indicates a need for large-scale analysis of third-party libraries in Android TV and other platforms to complement existing repositories.

We have shown that TV apps collect less sensitive data than their mobile counterpart. This difference might be because TV apps include fewer functionalities than mobile apps or because the Android TV platform needs more time to develop fully. It is worth noting that top streaming services such as Netflix and Disney+ moved away from the subscription-only revenue model to include advertising to their platforms[3]. This paradigm change could trigger ads spending and tracking services in Android TV and the mobile platform. The situation also suggests that the subscription model is insufficient for big companies. Thus, it remains to be seen the impact of this situation on streaming platforms in Android and the user's privacy.

5.7.2 Data Collection Practices

Our results show that TV apps rely heavily on static identifiers. Google recommends the class `AdvertisingIdInfo` for tracking, as it allows users to easily reset the value [169]. The fact that we found few resettable identifiers means that the policy of restricting advertising IDs [172] will have no real impact on the Smart TV ecosystem. In contrast, we found a prevalence of static identifiers, such as GUIDs, and hardware addresses.

Libraries are responsible for around 60% of all sensitive flows transporting identifiers. These flows are present in 85% of the TV apps. In our analysis, we detected that around 90% of the TV apps include advertisement or analytics libraries, some of them with a bad history of exposing sensitive data, such as Umeng, Appodeal, and Unity. What is worse, it is well known that many developers are not aware of the data collection practices by TPL, which poses an even greater threat to end users [19, 22, 144].

Our dynamic analysis experiments confirmed that many TV apps use static identifiers to track users and that these IDs are shared with tracker domains. We detected static identifiers in the network traces by comparing IDs found across different apps. However, our analysis cannot identify all dynamic IDs generated by different tracking SDKs.

A noteworthy example is the TV companion app `xyz.klinker.messenger` that sends texts/-multimedia messages. This app claims to be completely free, with no ads and no personal data collection in its Play Store description. However, it requests permissions to access the geolocation, user profile, and custom permissions from vendors like Huawei and Sony. Moreover, the app's privacy policy mentions collecting personal data (in contradiction with its description) such as social media, geolocation, OS information, and the list of third-party providers. We also noted that this app requests all dangerous permissions immediately after launch, which is against the best practices [173] because it obscures the association of run-time permissions with a specific action. We also reported this app to Google.

[3] Ads were added to Netflix in November 2022 and in Disney+ in December 2022 [171]

5.7.3 Limitations

Ecosystem. Our dataset is limited to around 4.5k unique packages. This is larger than previous studies of Smart TVs [28,29]. However, the heuristic used to search TV apps might have missed part of the ecosystem. Particularly, TV apps from underrepresented regions could be included by searching in other markets. Self-signed certificates provide limited guarantees about developer information, but it is still the most reliable approach to identify developers at a large scale [155]. Alternative approaches rely on fingerprinting and Machine Learning [174,175]. These techniques can complement our approach based on certificate fingerprint and custom permission.

Permissions. We only consider static definitions when analysing custom permissions. Omitting dynamic definitions can generate incomplete results. However, a previous study showed that only 3% of custom permissions are declared dynamically [159], making our results representative.

Third-Party Libraries The accuracy of the TPL detection tools depends on the completeness of library profiles. Additionally, these tools present some limitations when dealing with problems such as package flattening and dead code elimination. In contrast to the third-part frameworks, our custom approach described in Section 5.4.1 is not resilient to obfuscated package names, but it still allows us to detect many common libraries, including some libraries showing privacy-invasive behaviour.

Behavioural Analysis. Static analysis abstracts from user inputs and approximate runtime values, which can be a source of false positives. Additionally, our static analysis does not consider dynamic code loading, some cases of reflection, and native code. We detected that 6% of the apps in our dataset load dynamic code, while 60% include the required classes to do it. Similarly to `WearFlow`, a more precise evaluation of the Android TV ecosystem could add other static analysis frameworks described in Section 2.3.3 to the pipeline. While we use dynamic analysis to overcome these limitations, our traffic analysis experiment does not scale, mainly because we decided to log in on each app during testing to get more accurate results and the time limit of 15 minutes.

5.8 Related Work

There are several works that have studied different aspects of the mobile ecosystem. Wang *et al.* presented a large-scale analysis of the Play Store, where they discovered a 25% increase in ad libraries over a period of three years and a decrease in paid apps [52]. Similarly, other studies analysed Chinese markets [23], mobile ecosystem in general including TPL [17–21], tracking and advertising ecosystem [17, 18, 22–24], and trust in the Android ecosystem [154, 155, 174]. Although similar, the Android TV ecosystem has its peculiarities and has not been the focus of previous research to the same degree.

Several studies have used static analysis to detect potential information leaks and harmful behaviour in Android apps [32–34, 39]. Due to their popularity as a communication method, there have been previous research efforts looking at sockets to detect vulnerabilities and data

leakage [127–129]. We consider both approaches to analyse potential information leaks and vulnerable communication in TV apps.

Previous works reported vulnerabilities and data leakage in IoT devices, including Smart TVs and Chromecast devices [2–4, 13, 14]. Recently, two papers investigated security and privacy issues on Roku and Fire TV using traffic analysis [28,29]. They found that a large number of TV apps expose PII to third-party domains. They reported mixed results of DNS-based blocklists to prevent data leakage, with concerning results on static identifier leaks. Our work presents several differences. First, we use static analysis to detect potential data leaks and then we use the results to select target apps and sources for the dynamic analysis. We also focus on the Android TV ecosystem, which was only briefly discussed in Varmarken *et al.* [29]. Additionally, We present a comparison between apps on different platforms (TV vs Mobile). This analysis shows notable differences in terms of data collection practices and uncovered limitations of TV apps and Android TV.

Finally, other studies in this area have also analysed several attacks on Smart TVs [148, 149], review of privacy and security aspects [176], TV apps traffic fingerprinting [177], remote control mimicry [157] and user expectations regarding privacy [30]. Our paper complements these works by analysing several issues regarding TV app certificates and permission misuse. Some of these problems were previously reported in the mobile ecosystem [145, 158, 159, 178], but other problems manifest only on TV apps, such as inconsistent permission and missing functionalities. These problems emerge when developers port mobile apps to Android TV, doing minimal work to ensure the quality of their apps and the user experience.

5.9 Chapter Summary

In this chapter, we have presented a systematic study of the Android TV ecosystem. The analysis of more than 4.5k TV apps reveals pervasive sensitive data collection and tracking practices RQ1. Notably, we detected the prevalence of static identifiers over identifiers designed to protect users' privacy. Consequently, policies such as limiting access to advertising identifiers will have almost no effect on TV apps. Our measurements show that tracking and advertising services collect static identifiers and personal data in many cases, including TV-specific data.

While our static and dynamic analyses show that TV apps have fewer trackers than mobile apps, our work also sheds light on the developer ecosystem detecting multiple bad development practices that leave TV customers vulnerable. Moreover, developers porting mobile apps to Android TV produced inconsistent apps with much lower quality, including popular streaming services. We have shown that third-party library repositories are incomplete for TV apps. These are examples of platform-specific problems in Android TV RQ2. In the future, we hope the research community will give more attention to platform-specific problems in Android.

Overall, this chapter contributes to the understanding of the Android TV ecosystem and closes the research gap concerning the study of the mobile ecosystem. By studying multiple aspects of Android TV, we propose several practices that can improve this platform. In particular, we

encourage the development of better guidelines to migrate mobile apps to Android TV and the secure usage of new APIs for inter-device communication. Finally, Android TV should improve its UI, allowing users to check permissions and read privacy policies before installing apps on their Smart TVs.

6 Extracting Security Specifications from Software Documentation

In this chapter, we propose a novel approach that uses API documentation to represent Android methods in high-dimensional embeddings. The distributed embeddings encode semantic information that enables the inference of security and semantic properties that can be used to generate taint specifications and perform semantic search operations for different purposes. This removes the need to analyse code or perform feature extraction to detect security-sensitive methods.

6.1 Introduction

In the previous chapters, we conducted the analysis of information flows in wearable and TV apps. In both cases, we manually created taint specifications to run the experiments. This approach is prone to errors and does not scale. Moreover, security practitioners routinely use static analysis to detect other security problems in Android apps, as discussed in Section 2.3. The soundness of these analyses depends on how the platform is modelled and on the list of sensitive methods. Collecting these methods often becomes impractical given the number of methods available, the pace at which the Android framework is updated, and the proprietary libraries that Google releases on each new version.

Android releases include documentation that describes the correct use of its APIs, how complex components interact and general guidelines. A vast software project such as Android requires good documentation to simplify the app's development lifecycle. Considering the ever-changing Android framework, API documentation adds extra value by helping developers to cope with the constant evolution of the Android framework and Google libraries (even when some of these APIs are offered by proprietary libraries). Thus, Android documentation is an excellent source to infer semantic properties of the underlying implementation across releases.

This chapter presents `DocFlow`, a framework that uses NLP to capture API methods semantics directly from the documentation and can be used to generate taint specification for arbitrary platforms. Our framework implements a full pipeline, including crawling and parsing the documentation, classification of sensitive methods, and other semantic tasks. We use `DocFlow` to detect sensitive methods for the Android API version 32, and Google proprietary libraries, including the wearable and TV libraries. Moreover, we show that `DocFlow` can detect sensitive methods on each API release and different versions methods based on its semantics. We compare our results with other tools and find that `DocFlow` achieves comparable or better results. In summary, the contributions of this chapter are:

- We demonstrate that API documentation in well-documented libraries can be reliably used for auto-identification of sensitive methods and classify them into semantic categories.
- We present `DocFlow`, an end-to-end automatic tool that uses NLP to classify sensitive methods and perform semantic search operations.
- We compare our approach to published and well-cited baselines and significantly outperform them. Unlike the baselines, we do this without access to the source code of the Android framework or Google libraries. Moreover, our approach incorporates clustering and semantic search capabilities whereas the baselines only include classification.
- We show that our approach is robust against evolution of the library and its documentation by consistently showing accurate classification results across multiple versions of software documentation.

6.1.1 Motivation Example

The documentation of the Android framework and Google libraries is available in *Javadoc* format (in the source code) and on the developer's website. In Javadoc, the documentation of a method describes the purpose, the inputs and outputs, and sometimes other considerations such as permissions or performance. We revisit the data leak example from Section 2.2.1 and discuss how we can use documentation to produce taint specifications.

In Section 2.2.1, we described how app developers use source methods to access to shared resources and sink methods to expose data outside the app's context. The following code snippet in Listing 6.1 uses the `getLastKnownLocation` source and `writeFile` and `post` sinks. A security analyst must produce a complete list of sources and sinks to generate sound taint specifications and avoid missing sensitive information flows, i.e., one source and two sinks are required for our example.

```
1  public void onCreate(Url url, String filePath){
2      location = LocationManager.getLastKnownLocation()  // source
3      ...
4      //some code
5      ...
6      FileOutputStream file;
7      file = new FileOutputStream("file_name");
8      file.writeFile(filePath,location);  //sink 1
9
10     HttpURLConnection connection;
11     connection = new HttpURLConnection(url);
12     connection.post(url, location);  // sink 2
13 }
```

Listing 6.1: Source and sinks example. A more detailed explanation about sensitive APIs can be found in Section 2.2.1

Consider the `getLastKnownLocation` documentation. Figure 6.1 shows the Javadoc of this method in the Android website. The online documentation is generated from the Javadoc description in the method source code. We can observe the method signature followed by a description and details about the parameters and return type. The full signature includes the access modifier (public), the return type (Location), the method name (getLastKnowLocation), and the parameters (String provider).

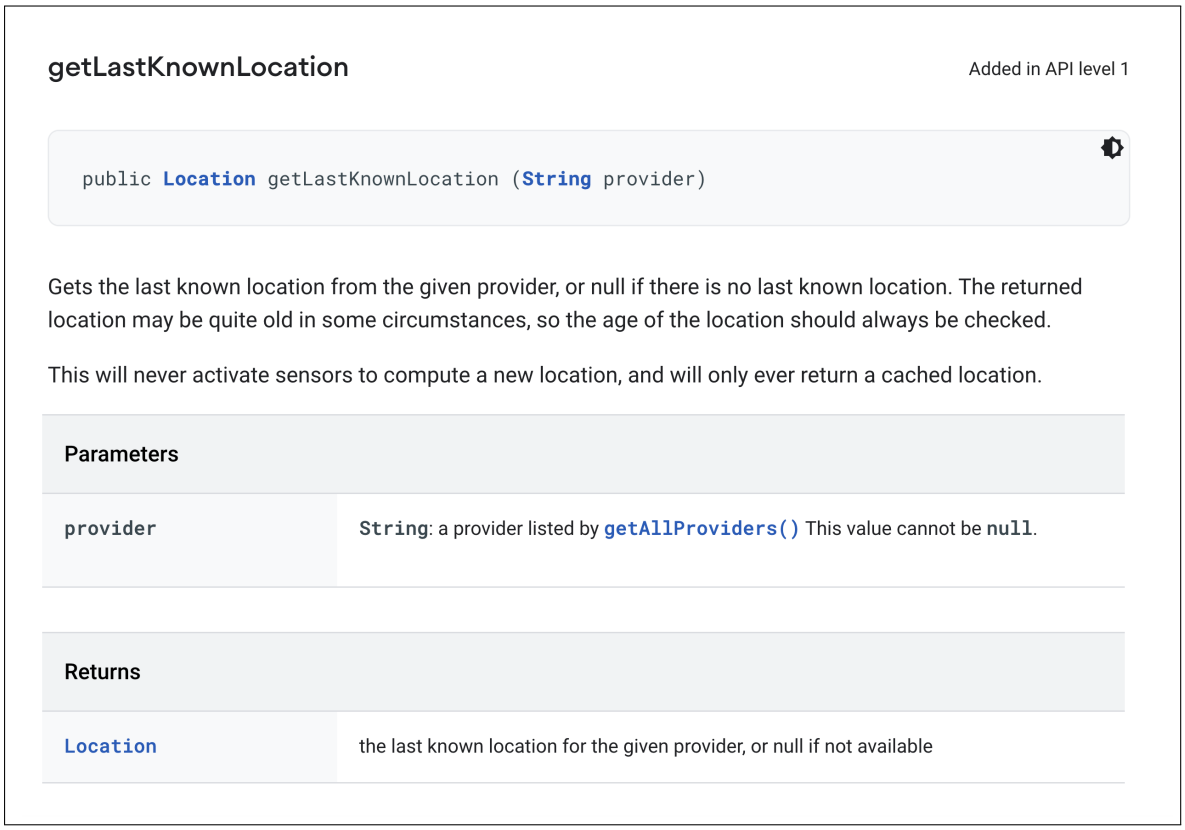


Figure 6.1: `getLastKnowLocation()` method documentation

Consider now the documentation corresponding to the `LocationManager` in Figure 6.2. This class declares the `getLastKnowLocation` method used in the example above (Listing 6.1). The class description is complemented by permissions details and class hierarchy. Looking at the documentation of this method and its corresponding class, we can infer that this method is a source. The class documentation describes the location services it provides, and the method documentation describes what type of location information is retrieved. This example also shows the rich semantic information that class and method names can provide if good development practices are in place. We explore how this information affects `DocFlow` in Section 6.3.

Suppose a security analyst needs to evaluate APKs compiled for different Android versions or a specific platform. In that case, the analyst must produce taint specifications for each Android platform and all relevant Google libraries. As researchers found that apps tend to have more sensitive data leaks with newer Android releases [60], complete taint specifications are critical to avoid missing data leaks.

The community have proposed program analysis and language specification techniques to find

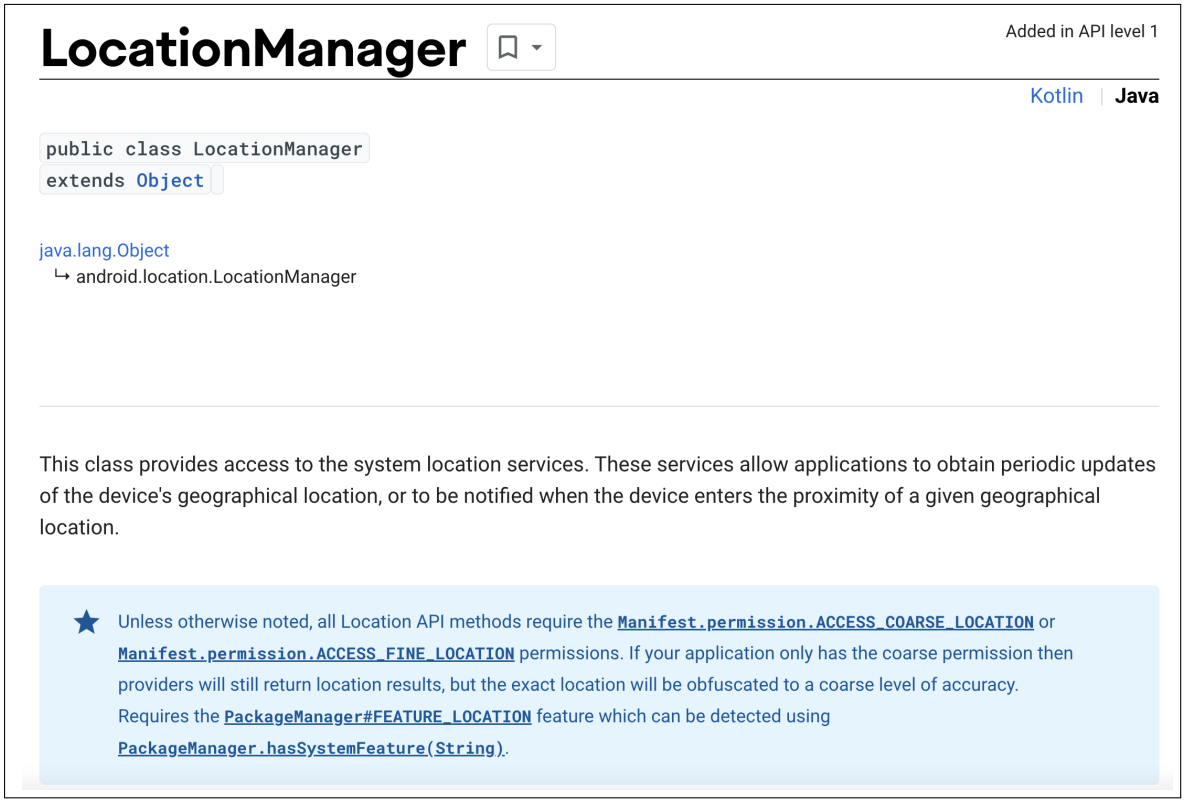


Figure 6.2: LocationManager class description

candidate methods from a large collection of Android methods [51, 62–65]. However, these techniques present limitations when dealing with obfuscation tools, which raise the complexity of automated code analysis [66], and closed-source libraries. For instance, Antonioli *et al.* had to reverse engineer the Google Nearby Connection API and develop a custom implementation for evaluation [54]. The techniques mentioned above face similar problems as the code is unavailable for inspection. In this context, we propose a novel approach to detect sensitive methods in Android using documentation without resorting to program code.

Last, we want to highlight that software documentation is a rich source of information for learning about programs and it is instrumental to the success of software that relies on APIs and follow good documentation practices [53] [1] At the same time, these good practices enable complex inferences. For instance, Chaudhary *et al.* developed a compiler extension that searches bugs based on rules extracted from document specification [179]. Similarly, Gorla *et al.* identified several abnormalities by checking apps' behaviour against their description [104]. Overall, Android documentation is a great source for extracting syntactic and semantic information about API methods and classes.

Location package evolution

Before presenting `DocFlow`, we continue by showing the evolution of the *android.Location* package. Note that we do not pretend to study the evolution of Android APIs, as this has been

[1] We refer the reader back to Section 2.4 for a discussion on good documentation practices and API usability.

done before [53, 59, 93, 180]. Instead, we want to give a notion of how this package evolves, as we have used it in our illustrative example (Listing 6.1) and we continue using this package in our evaluation.

Table 6.1 shows the evolution of this package from API level 29 to 32. The values in column NM (new methods) indicate that methods are constantly added on each release. The following three columns indicate how recurrent methods change across API levels. For instance, looking at the API 29 → 30 update, 18 % of the classes description differ (C), 14% of method descriptions present changes (M), of which 30% are due to methods being deprecated (D). These numbers indicate that the documentation of recurrent methods is also subject to modifications, and only a portion of these changes are due to deprecation. Figure 6.3 illustrates the difference in the `LocationManager` class description for the API upgrade 29-30.

API Change	NM	C	M	D
29 → 30	14	0.18	0.14	0.30
30 → 31	21	0.18	0.25	0.16
31 → 32	9	0.35	0.34	0.05

Table 6.1: Location package evolution. (29 → 30) indicates the update from API level 29 to 30. NM stands for New Methods (total). The other columns represent percentages: C (classes modified), M (methods modified) and D (methods modified by deprecation).

```
Android 29:
"This class provides access to the system location services.
These services allow applications to obtain periodic updates of
the device's geographical location, or to fire an application-
specified Intent when the device enters the proximity of a given
geographical location."

Android 30:
"This class provides access to the system location services.
These services allow applications to obtain periodic updates of
the device's geographical location, or to be notified when the
device enters the proximity of a given geographical location."
```

Figure 6.3: LocationManager class description for API levels 29 and 30.

These results align with a recent work [180] that studied changes in code and API documentation across versions, and other works that analysed the evolution of API documentation [53, 93]. Despite this constant evolution, app developers cope with all these new features thanks to the documentation that comes with each new Android release. Our solution to detect sensitive methods takes advantage of this fact and models the semantics of API methods using their documentation. We present `DocFlow` below.

6.2 DocFlow Overview

In this section, we describe `DocFlow`, a system that leverages documentation to generate sink and source definitions for the Android platform. `DocFlow` takes the description of an API

method found online and generates a *document* that is then used to classify and extract semantic information about that method. Note that we use the term *document* as normally used in the NLP literature. We generate *documents* from code documentation, but the two concepts are not directly related. We first present our design principles, and then the `DocFlow` implementation details.

6.2.1 Design Principles

`DocFlow` uses NLP techniques to detect sensitive methods using Android documentation. In particular, our methodology is based on the following observations:

Maturity The Android source code is well documented. All methods from the Application and Java frameworks have their corresponding documentation. Android AOSP and Google Play Services developers do not accept commits without a proper code style including the Javadoc of every non-trivial method. We provide more details when describing the dataset in the next section.

Structure Android developers follow a convention when writing documentation. For instance, every class must have a description and methods documentation should start with a third-person descriptive verb. These guidelines allow us to learn patterns from Android Javadoc. The style guide for contributors is described in Android's website [2].

Semantics A method's documentation reflects the contract between the caller and callee methods. An API is defined by the combination of the caller interface and the description of the usage protocol (signature) and semantics [93], while the implementation details are hidden in the callee method. In general, the documentation specifies the purpose of a method, parameters, and return type. Additionally, a class description includes the purpose of the class and the abstraction it provides.

These properties make Android a suitable platform to automate method classification via documentation analysis (instead of a code-based approach). All this documentation provides rich semantic context that can be used to make natural language inferences. `DocFlow` uses word embedding techniques (Section 2.4) to calculate the distributed vector representation [99] of Android methods. While our framework is suitable to perform classification and search tasks for sentences in general, we focus our effort on detecting sources and sinks for static taint specification in Android. We explain our methodology below.

6.2.2 Docflow Methodology

Figure 6.4 presents an overview of `DocFlow`. We briefly describe the overall approach, and then we go through each step in detail. First, our framework crawls the Android and Google Play Service websites to download and parse the method's documentation. The second step sanitises

[2] https://source.android.com/docs/setup/contribute/code-style

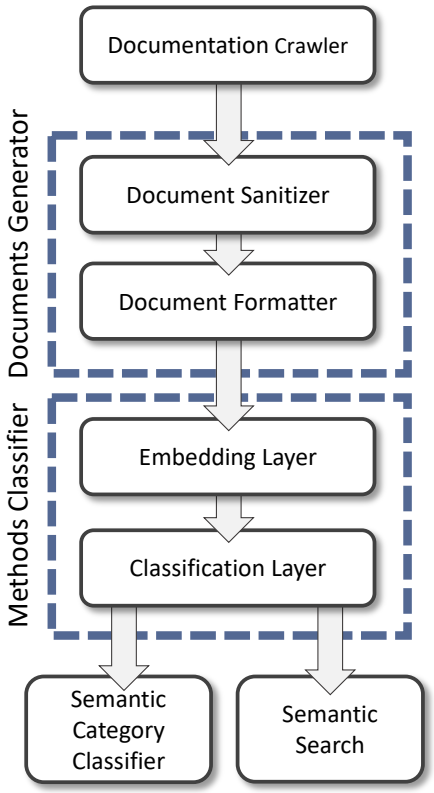


Figure 6.4: `DocFlow` overview.

the raw data and generates different documents (input format) for each method. These differ in the amount of information that describes each method and will be used to find the best representation. The Method Classifier uses a Transfer Learning [181] approach and consists of two layers. The first layer calculates the vector embedding using a language model and transforms each document into a high-dimensional distributed vector. The second layer connects the embedding layer to a classification layer that separates security-sensitive and non-sensitive methods. Finally, a Semantic Classifier module further classifies methods into semantic categories, e.g., file system or network connection. `DocFlow` also includes a Semantic Search module that reuses the pre-calculated embeddings from the method classifier to search for methods that are close in the vector space using metrics such as cosine similarity. With this pipeline, `DocFlow` relieves security analysts of much of the substantial manual effort of generating taint specifications. In the following sections we describe each of `DocFlow` components in detail.

Crawler

`DocFlow` downloads the reference section of the Android developers and Google Play Services websites[3]. The first step consists of extracting all the Android packages from the website map. Then, `DocFlow` iterates over all packages, classes, nested classes, interfaces, and methods to extract the documentation. For each method, we extract the signature and description. The signature includes the class name, method name, return type and parameters. Additionally, we

[3] https://developer.android.com/reference/packages
https://developers.google.com/android/reference/packages

download the description of all classes. All this information is stored in JSON format. The next step processes the JSON files to generate the *document* for each method.

Documents Generator

A *document* is the text representation of each API method. This module cleans the data from the JSON files and then generates different document formats. The raw data from the files contain special characters, numbers, and sometimes HTML elements. All these characters must be cleaned before generating the method representation and passing the document to the embedding layer for further processing. For this purpose, the `DocFlow` formatter module implements the following three steps:

1. Sift and Uniquify. We remove methods with short description, e.g., less than 20 characters or 3 tokens. We also remove duplicated methods to avoid the duplication effect [182] that could inflate the performance of the models. We considered two methods duplicated if they have the same method name and description. The Android framework contains many classes that offer the same functionality across different packages. We avoid this case by removing duplicates. Note that this does not considering the same method with different signatures.

2. Sub-tokenise. We sanitise tokens for rare words, e.g, camel case tokens [183], and tokens with special characters (.,$, or numbers). NLP models are unlikely to produce good embedding for tokens unseen during training or tokens with very low occurrence. Consider the following method description: "Returns the `serverClientId` that was set in the request". The meaning of the token `serverClientId` might be clear for a human reader, but an NLP model will benefit from splitting this token into three sub-tokens (server, client, id). This transformation takes more relevance because of the Android source code writing style.

3. Deprecation-check. We remove all deprecated documents. These contain the deprecated annotation or the word deprecated in the description. While these methods may still be used, their documentation is no longer maintained under the same principles as non-deprecated methods, so they normally contain non-relevant descriptions or point to the newer alternative.

`DocFlow` generates multiple formats for each document to find the best representation. We aim to use the best representation for a particular classification or clustering task. In the evaluation (Section 6.3) we explore the effect of using multiple granularities to generate the documents. For instance, consider the `getLastKnownLocation()` documentation from Figure 6.1 and its class description from Figure 6.2. An illustrative example of two potential formats is shown in Figure 6.5. A simple format (1) uses only the method description, while a more complex representation (2) includes the corresponding class description.

Methods Classifier

The sources-sinks classifier consists of an *encoding step* and a *classification step*. First, the encoding step calculates the embedding representation of each method using the Sentence-BERT

Method: getLaskKnowLocation()

Potential representations:

(1): Gets the last known location from the given provider, or null if there is no last known location. The returned location may be quite old in some circumstances, so the age of the location should always be checked.

(2): Gets the last known location from the given provider, or null if there is no last known location. The returned location may be quite old in some circumstances, so the age of the location should always be checked. This class provides access to the system location services. These services allow applications to obtain periodic updates of the device's geographical location, or to be notified when the device enters the proximity of a given geographical location.

Figure 6.5: *getLastKnownLocation* method representations (document formats): (1) method description. (2) method description + class description. Formats (1) and (2) correspond to the formats (A) and (E) in the evaluation section.

network. This model achieved state-of-the-art performance over several NLP task benchmarks [102] and was described in Section 2.4.2. The input of the embedding layer is the document generated in the previous step, and the output is a vector embedding of 768 dimensions. This embedding is suitable for semantic operations or downstream tasks, and we use them for both cases: classification and similarity search tasks. Note that `DocFlow` is flexible regarding the specific embedding model. We further discuss the model choice in Section 6.4.

We propose an optional fine-tuning step in the embedding layer to better separate sources/sinks methods in the vector space. For this, we adopt a contrastive learning [184] approach and the STS task. We use the inference architecture, as in Figure 2.13, and a dataset composed of pairs of methods and a similarity score to continue tuning the model. The aim of this step is to minimise the distance between semantically similar sources/sinks and maximise the distance between unrelated methods.

The second step consists of a feature-based classifier that separates sources and sinks from other methods. We connect the embedding layer with a customisable classifier where the final decision is up to the analyst. This step and the fine tuning require a dataset with annotated labels, .e.g, Android methods with a source, sink, or neither label (See Section 6.3 for details on the dataset we used). Note that our approach easily generalises to classify other method types. The only requirement is to produce a training set with the corresponding labels and (optionally) fine-tune the embedding layer. We want to emphasise that `DocFlow` is flexible regarding the specific model choice for the Embedding and Classification layers.

Semantic Category Classifier

This module further assigns a semantic category to each method such as Network, Files, and Identifiers. This division is useful in case the analyst is only interested in a subset of method such as network connection APIs. For this second classification task, we use an unsupervised classifier based on a zero-shot approach [185] that does not require a labelled dataset of Android methods with their semantic categories for training.

The classification problem is modelled as a text entailment problem [186] where the task is to decide if the premise entails the hypothesis. Figure 6.6 shows an overview of this approach. The document to be classified (*doc_x*) is used as the premise, and each label (file, location, and network) is converted into a hypothesis, e.g., "*is doc_x* about location?". Then, the embedding layer encodes the pair (Pair encoder) and passes the output to a sequence-pair classifier that predicts if the premise entails, contradicts, or is neutral to the hypothesis. The probabilities for entailment are converted into category probabilities, and we finally take the probability of entailment as the probability of the label being true. For instance, if we want to classify the method `getLastKnowLocation`, the classifier first constructs one hypothesis for each possible category (e.g., geolocation, file system), then it passes to the sequence-pair classifier, and selects the label with higher probability as the category for the method.

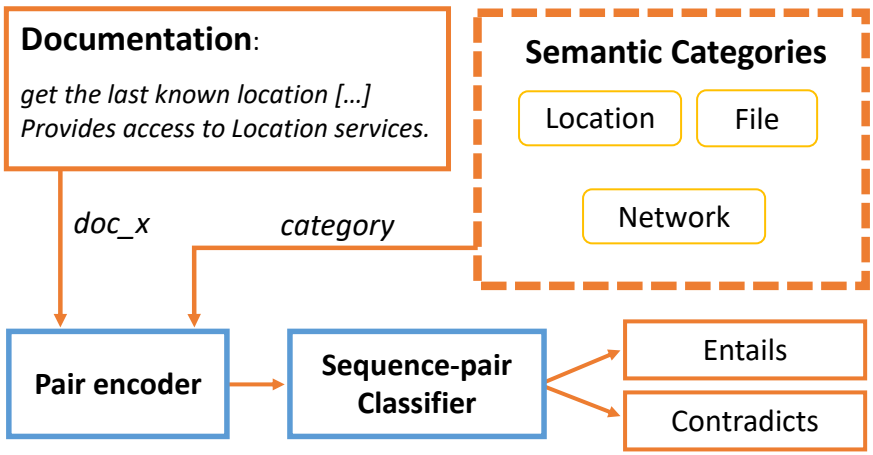


Figure 6.6: Semantic Category Classifier Overview.

Note that this step is independent of the Method Classifier module. Although in practice, it might be useful to run after detecting sources and sinks. This approach enforces label and document understanding, and does not rely on task-specific training, so previous observations of labels are not required. The `DocFlow` zero-shot classifier is flexible regarding labels, i.e., the user only needs to add or remove categories from the list of labels. Last, this module is also flexible regarding the sequence-pair classifier. In our evaluation we use a pre-trained model fine-tuned with the MultiNLI dataset [187], but the decision is up to the user. This classification module can reuse the embeddings calculated in the previous step (for sources and sinks detection) if available.

Semantic Search

The Semantic Search module solves the problem of finding semantically similar methods. This step reuses the embeddings calculated in the Methods Classifier module (it can also calculate the embeddings from scratch). Figure 6.7 illustrates our approach for a symmetric semantic search. In this setting, the query and corpus entries are roughly the same length. The module first embeds all documents from the dataset. Then, a query document is embedded into the same vector space, and the closest documents are found using a similarity metric. Overall, the embedding layer derives semantically meaningful vectors that can be compared using several similarity measures, e.g., cosine similarity and euclidean distance. This property allows us to use the embeddings for semantic search and clustering efficiently.

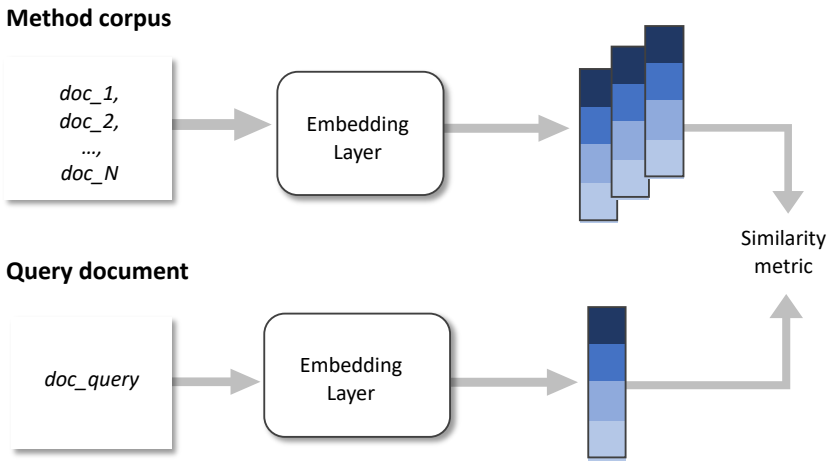


Figure 6.7: Semantic Search module overview

`DocFlow`'s Semantic Search module allows an analyst to easily adapt taint specifications to new API levels or library updates. A new API release might add new versions of sensitive method (different signature or wrappers), or new methods that offers similar functionality for deprecated APIs. Similarly, the Search module can be used to complement a taint specification, i.e., by searching similar methods from an initial list of sources and sinks. Overall, our semantic search and classification modules can help to automatically detect sensitive methods in all these cases. Therefore, `DocFlow` is an excellent tool for generating sound taint specifications and alleviate the process of manual selection. The Android framework and platform libraries use the same documentation approach. Thus, `DocFlow` is suitable for both cases, as we show in the evaluation below.

6.3 Evaluation

We start by explaining the setup for the experiments. Our evaluation first compares the performance of the document formats and classifiers. Second, we compare `DocFlow` with baseline tools that detect sensitive methods. Third, we evaluate the robustness against software evolution and partial specifications. Last, we show the evaluation of the semantic classifier.

6.3.1 Experimental Setup

We first provide details of the dataset followed by the formats we use to represent APIs, their neural embeddings and the classifiers. We evaluate `DocFlow` with a corpus of Android AOSP and Google Play documentation obtained in March 2022. This corresponds to the API level 32 and Android version 12. The scripts and dataset are available in `DocFlow`'s repository (Section 1.4).

Dataset. We use `DocFlow`'s crawler to download a corpus of 46227 methods. Android AOSP methods form the bulk of this dataset with 37692 methods from 4538 classes. All framework methods contain a description, and 2% correspond to deprecated methods. Regarding Google Play Services, we collected 8535 methods from 1191 classes, of which 80% include a description and 4% are deprecated methods. A further 15% corresponds to methods from deprecated classes. For the remaining cases, we use the method name as the description.

Labels. We rely on mix of expert hand labelling and data from other repositories to get the ground truth for our dataset used during the evaluation. In particular, we use the following sources: 1) Labelled wearable APIs from Chapter 4, and TV APIs from Chapter 5. 2) We randomly selected 3K methods from baseline tools 6.3.3 to train the classifiers (neither 43%, sources 35%, and sinks 21%). Note that, in some cases, these methods contain the predicted label but not the ground truth. 3) A disjoint group of 96 Android methods with labels for the fine-tuning step from baseline tools. 4) The *android.Location* package and four Google Play Service libraries.

We manually annotate each method (without label) with one of three categories: source, sink, or neither. The methods were independently labelled by three members of the `S3Lab` from Royal Holloway to ensure the quality of the labels. We proceed to label all methods and then discussed complex cases until reaching an agreement. The initial phase produced a matching of 80% of the labels. All the mismatches involve source-neither or sink-neither conflicts. There were no confusions between source-sink labels. Still, the fact that around 20% of the methods were assigned with different labels shows the limitations of manually collecting sensitive methods, even when the volume of methods is small. We discuss the limitations of our approach in Section 6.4.

Representations. We generated four document formats to evaluate the source-sink classifier. Table 6.2 shows how each format is constructed. Format A is the simplest case where only the method description is used. Each format adds more information to the method representation, such as class description and return types. Format D is the format that includes more information (includes the full method signature, method description, and class description). Multiple formats allow us to understand what parts of the documentation encode better information for each task.

We use different formats for the semantic category classification. Table 6.3 shows these document formats. We emphasise class documentation (instead of method description or signature) because they are a more natural choice to encode information about categories.

Format	Method representation (document)
A	method description
B	method name + description
C	method signature + description
D	method signature + description + class description

Table 6.2: Document formats used for the source-sink classification

Format	Method Representation (document)
E	description + class description
F	class description
G	method name + class description

Table 6.3: Document formats used for the semantic categories classification

Embeddings. For the Embedding Layer, we use the pre-trained Sentence-BERT model `all-mpnet-base-v2`, which has the best overall performance[4]. We use a set of hand-annotated methods for fine-tuning this model. First, we generate pairs of methods with a similarity score (8k pairs). The similarity score is automatically calculated based on the category (source, sink, neither), and the cosine similarity between the pairs. Finally, we fine-tune the model with the following configurations: 10% of the training data for warm-up, 4 epochs, batch size of 16, cosine similarity as loss function, and we evaluate every 1000 steps against another set of hand-annotated methods with their corresponding score. We tested with different settings recommended by the Sentence-BERT authors and selected the one with the best performance.

Classification. To evaluate `DocFlow` performance, we split our dataset into training (75%) and test (25%) sets, and we use four classifiers: Logistic regression, SVM, XGboost, and a 1-layer neural network. These classifiers have shown to produce good results with high dimensional embeddings [51, 101, 102, 108]. We evaluate the performance of the classifiers using the following metrics: accuracy, precision, recall and F1-score. We use the cosine similarity metric for semantic search operations. These metrics were defined in Section 2.4.2.

6.3.2 Efficacy of Representations

We first evaluate the impact of the four document formats on the precision of the classifiers. Figure 6.8 illustrates the performance of the classifiers with the input formats. The results indicate that the precision increases as we add more information to the method representation. The simplest format (A) performs worst in all cases, and the most complex (D), which includes the full signature and class description, achieves the best performance. The difference is less significant between formats (B) and (C). The latter uses more information to represent the signature, but the descriptions remain equal. Overall, the results indicate that encoding more information in a document representation produce better embeddings to classify methods. We use Format D for the following experiments unless otherwise stated.

[4] https://www.sbert.net/docs/pretrained_models.html

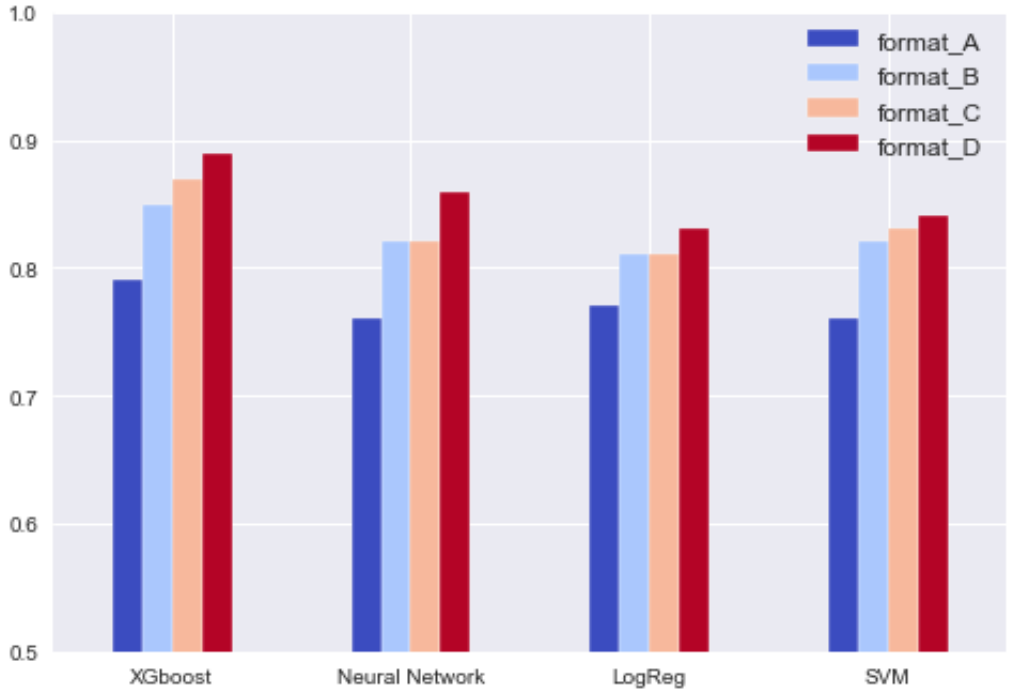


Figure 6.8: `DocFlow` precision using different document formats

Next, we explore how the performance of `DocFlow` changes using different classifiers. We use the four classifiers mentioned above and a custom classifier built on top of the Semantic Search module. For this, we use the embeddings from the training set to classify embeddings from the test set based on the cosine similarity. This approach assigns the label of the most similar method from the training set.

Table 6.4 reports `DocFlow`'s evaluation metrics using the format D and the fine-tuned Sentence-BERT model. The high precision and recall indicate that the models rarely make mistakes when predicting sources and sinks. While all classifiers achieve comparable results, XGboost achieves the best results considering all metrics. Therefore, we use XGboost as the default classifier for the remaining experiments. Even though the Semantic Search approach achieves the worst results, the performance is still comparable to the other classifiers using a relatively simple approach.

	Accuracy	**Precision**	**Recall**	**F1-score**
LR	0.86	0.85	0.85	0.85
SVM	0.85	0.84	0.84	0.84
XGboost	**0.89**	**0.88**	**0.87**	**0.87**
NN	0.86	0.85	0.85	0.85
S-Search	0.82	0.80	0.79	0.79

Table 6.4: `DocFlow` performance using a Fine-tuned Sentence-BERT in the embedding layer. Classifiers: LR(Logistic Regression), SVM (Support Vector Machine), NN (Neural Network), S-Search (Semantic Search)

Running `DocFlow` on our entire corpus produced a total of 11456 sources and 6367 sinks from 37692 framework methods from the API level 32. While `DocFlow` scales to analyse the entire

corpus, we use a subset[5] of the `android.Location` package to manually validate the results. We use this package because it is a popular API that contains many sensitive methods and has significant churn in its API. In total, these classes contain 195 unique methods. `DocFlow` detects sources with a precision of 90% and 89% recall. The precision and recall for sinks are 70% and 96% respectively.

We observe that many methods that register callbacks are classified as sinks, such as `addNmeaListener` and `registerGnssStatusCallback` (`LocationManager` class) resulting in a lower precision for sinks compared to sources. The dataset for this experiment has annotations for sources, sinks, and all other methods are grouped as neither. The fact that the "neither" group aggregates many method types prevents `DocFlow` from modelling these method types with better precision. We acknowledge that separating methods into more granular labels, such as callbacks and constructors, will produce better results. Moreover, we could have defined more granular labels for sources and sinks. All of these come at the cost of more work during the labelling.

We further use t-SNE [188] to project the embeddings of the *android.Location* package in a 2-dimensional plane. This technique stands out for producing visualisations that reveal structural information for high-dimensional data. Figure 6.9 shows the projection of sources and sinks embeddings. The projection shows that sources and sinks tend to be separated in the vector space allowing the classifier to distinguish these classes with high precision and recall. The embedding models are designed to produce similar vector representations for methods with similar semantics. Moreover, the fine-tuning step aims to make this distinction clearer. This property enables our classifiers and semantic search module to distinguish methods based on their semantics.

6.3.3 Baseline Comparison

We compare `DocFlow` results against SuSi [51] and SWAN [61]. Both tools use a similar approach to classify sources and sinks. First, they extract syntactic and semantic features from the source code and then train a Machine Learning classifier. Their difference resides in the feature extraction process. We use two settings for `DocFlow`'s Embedding Layer: the default (DF) setting uses the pre-trained Sentence-BERT, while the fine-tuned (FT) setting performs the optional fine-tuning step. Table 6.5 shows that `DocFlow` (DF) outperforms both tools in precision and recall. Moreover, we can see that the FT model improves the result by an extra 5%. Overall, fine-tuning the Embedding Layer consistently improves the results of pre-trained models for all classifiers and metrics.

Platform libraries. As we mentioned previously, the code for Google Play Services libraries is unavailable for analysis. Thus, frameworks that use program analysis techniques to detect sensitive methods cannot model these libraries properly. As `DocFlow` does not present this

[5]We use the following APIs (same as Table 6.1): `Location`, `LocationProvider`, `LocationManager`,`GnssAntennaInfo`, `PhaseCenterOffset`, `SphericalCorrections`, `GnssMeasurementsEvent`, `GnssMeasurementsEvent`, `GnssNavigationMessage`, `GpsSatellite`, and `GpsStatus`

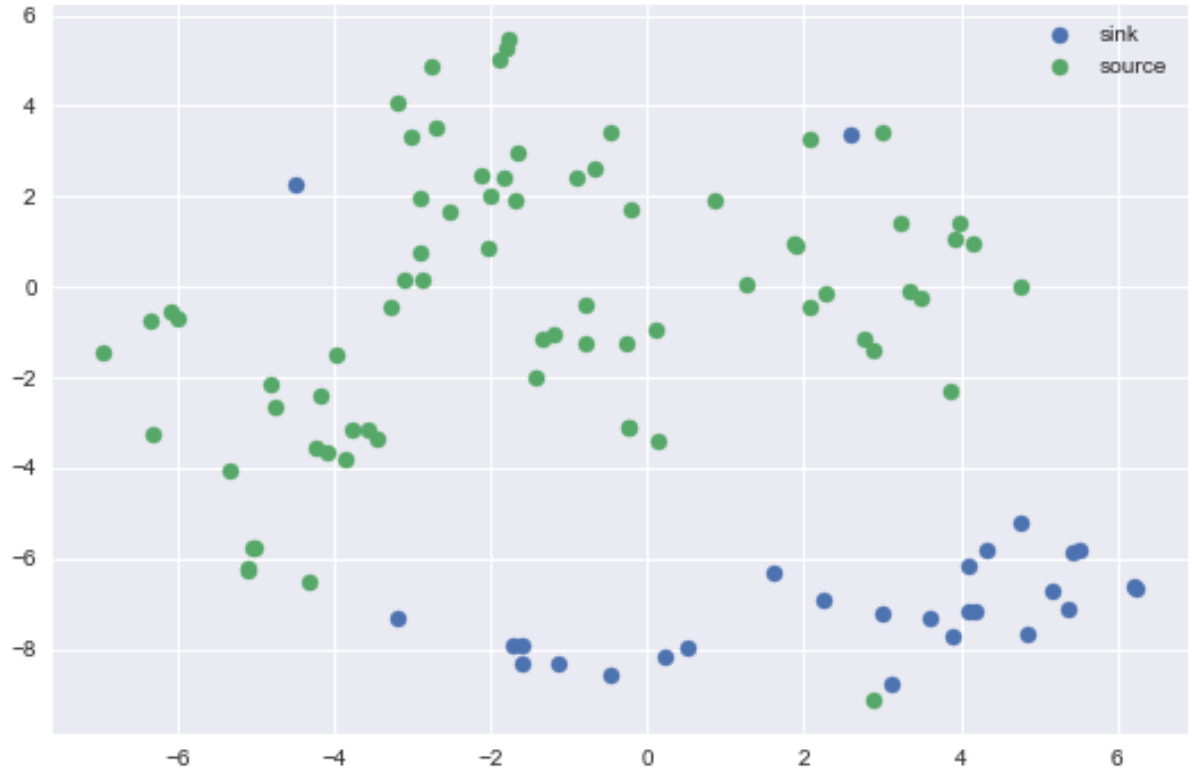


Figure 6.9: Visualisation of sources and sinks embeddings from the `android.Location` package projected in a 2D space.

	Accuracy	**Precision**	**Recall**	**F1-score**
SuSi	0.78	0.79	0.77	0.78
SWAN	0.89	0.76	0.67	0.71
`DocFlow` (DF)	0.84	0.83	0.82	0.83
`DocFlow` (FT)	**0.89**	**0.88**	**0.87**	**0.87**

Table 6.5: `DocFlow` vs SuSi comparison. (DF) default pre-trained Sentence-BERT and (FT) the fine-tuned model

limitation, we run `DocFlow` classifier on all Google Play Services libraries and evaluate four of them (Wearable, TV, Analytic, and Ads). The process of generating the ground truth for these libraries was discussed in Section 6.3.1. In total, this dataset has 523 methods.

We first run `DocFlow` for all Google Play Services methods and then validate the results with the four libraries. In total, `DocFlow` detected 2787 sources and 2283 sinks from 8535 Google Play Services methods. Table 6.6 shows the percentage of sources and sinks detected on each Google Play Service library. `DocFlow` is able to detect most of the sources and sinks for these libraries. We further analysed the cases where `DocFlow` misclassified the methods, and found that these cases usually involve callbacks and constructors. This result reinforces the idea that `DocFlow`'s precision would increase if the semantics of other method types are available during training. We provide an example adding callback labels in Section6.3.5.

	Package	Sources	Sinks
Wearable	`com.google.android.gms.wearable`	0.95	0.96
TV	`com.google.android.gms.cast.tv`	0.92	1.00
Analytics	`com.google.android.gms.analytics`	1.00	0.93
Ads	`com.google.android.gms.ads`	0.92	0.85

Table 6.6: Sources and sinks detected by `DocFlow` per library

6.3.4 Robustness against Software Evolution

One motivation to develop `DocFlow` is to automatically generate taint specifications for new framework levels or Google libraries releases. In Section 6.1.1, we first looked at documentation changes across versions (API methods and classes). Now, we evaluate how well `DocFlow` performs across versions.

We run the `DocFlow` classifier (default settings) against the four API levels (29 to 32) of the subset of the `android.Location` package. Figure 6.10 shows the increase in correctly predicted sources and sinks per version. This result indicates that `DocFlow` correctly adapts to changes in the Android framework, while the missed sources and sinks remain almost constant across versions (reduced for sources). The increased number of sources and sinks aligns with the data shown in Table 6.1.

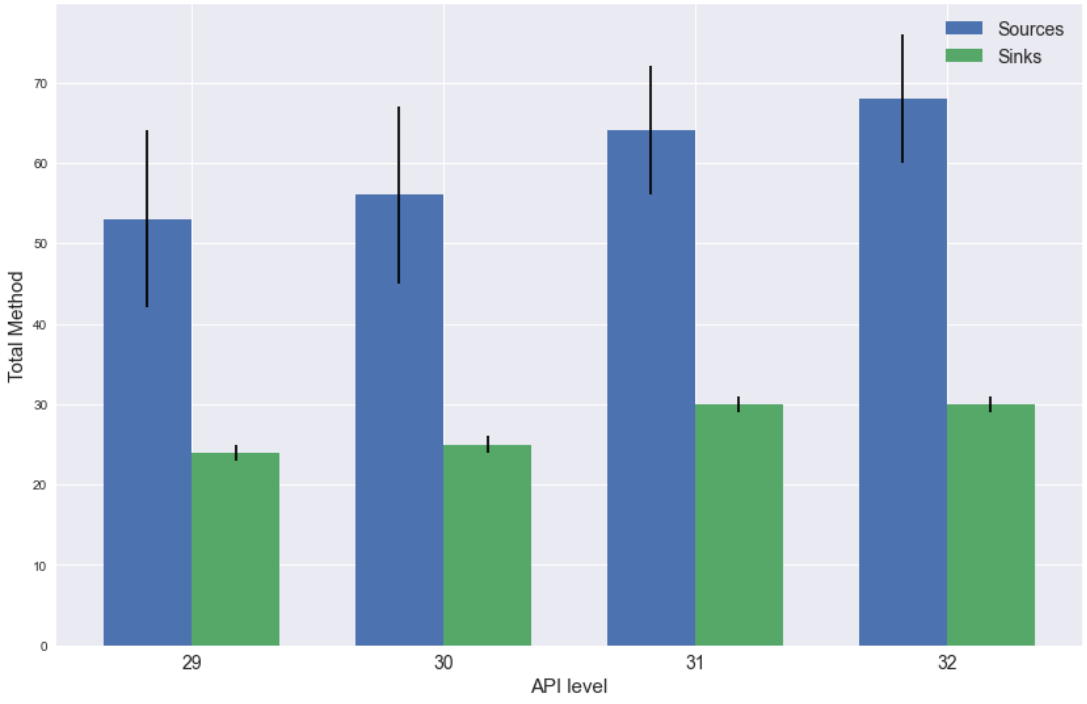


Figure 6.10: Number of detected sources and sinks across API levels. The sticks indicate the missed sources and sinks

We next project the source and sink embeddings from the four API levels in pairs. Figure 6.11a illustrates the sources and sink for API levels 29 and 30, while Figure 6.11b does it for APIs 31 and 32. These figures, complemented by manual analysis of individual points, shed light on how the Embedding Layer models the documents. First, it reinforces the observation that

sources and sinks tend to be separated in the vector space. Second, different versions of the same method (including across different releases) are close in the vector space. The points in pairs correspond to either the same method from different API levels or the same method with a different signature. Thus, `DocFlow` can detect sensitive methods across versions and successfully models changes in the documentation.

We further elaborate on how `DocFlow` models methods with different signatures or versions. For this, we use `DocFlow`'s Semantic Search module to cluster methods from two API levels and analyse the methods assigned to each group. We limit the subset of the *Location* package for readability to methods with different representations (format D) for API levels 31-32. Our clustering follows a community detection approach where the communities are formed by embeddings closer than a threshold (cosine similarity = 0.85).

Table 6.7 shows an extract of the communities corresponding to source methods. For most methods, both versions (31-32) were assigned to the same community. The communities 0 and 2 have 4 methods that correspond to the same method with different signatures. The community with id 3 is the only community that has two different methods, `getLatitud` and `getLongitud`. Still, these methods provide a similar functionality. Only three methods have their versions assigned to a different cluster[6]. This result provides further evidence that `DocFlow` is able to model variations in the documentation across versions.

Community	Method signature	Distinct Methods (n)
0	LocationManager_getCurrentLocation	4
1	Location_getAccuracy	2
2	LocationManager_getProviders	4
3	Location_getLatitude	2
3	Location_getLongitude	2
4	Location_getBearing	2
5	Location_getSpeed	2
6	Location_getElapsedRealtimeUncertaintyNanos	2
7	Location_getProvider	2
8	Location_getExtras	2
9	GnssMeasurementsEvent_getMeasurements	2
9	Location_bearingTo	2
10	Location_distanceTo	2
11	Location_getAltitude	2
12	Location_getTime	2

Table 6.7: Communities found using Format D and the Semantic Search module. The Method signature column follows the format class_method name

6.3.5 Partial Specifications

We evaluate the `DocFlow` Semantic Search module on the task of completing a partial taint specification using the dataset of wearable methods (266 methods with labels) from Google Play

[6] All methods are from the `Location` class: `getBearingAccuracyDegrees`, `getSpeedAccuracyMetersPerSecond`, and `getVerticalAccuracyMeters`

(a) Document embedding projection on a 2D space of the sources and sinks included within the Location package for API 29 and API 30

(b) Document embedding projection on a 2D space of the sources and sinks included within the Location package for API 31 and API 32

Services. In this case, we added the *callback* label to the previous set of labels (source, sink and neither). Callback methods are important as they allow precise call graph construction for

Android apps. We simulate the partial specification by randomly splitting our dataset into two halves: a corpus and a query dataset. This experiment aims to assign labels to the query dataset using the existing taint specification (corpus).

The semantic search takes each document from the query dataset and searches for the top three most similar methods in the corpus dataset. We use the cosine similarity as the metric and the fine-tuned Sentence-BERT encoder from previous experiments. We use two approaches for assigning labels: 1) the closest in the vector space (best-match). 2) The most common of the top three (top-3).

`DocFlow` achieves an overall precision and recall of 90% using the best-match approach with Format C (we tested with formats A-D). Table 6.8 shows the results per category (source, sink, callback) using these settings. In general, the results using the top-3 approach are worse than the best-match by a margin of 2-10%. This result show that `DocFlow` semantic search can be used to complement existing taint specifications, and `DocFlow` captures the semantics of different method types.

	Total	**Accuracy**	**Precision**	**Recall**	**F1-score**
Sources	101	0.97	0.92	0.94	0.93
Sinks	33	0.71	0.88	0.78	0.83
Callbacks	34	0.94	1.00	0.97	0.98

Table 6.8: Methods classification using semantic search and the best-match approach

6.3.6 Identification of Semantic Categories

We evaluate our Semantic Category classifier with a dataset of 358 methods and compare the results with SuSi category classifier (SWAN does not offer this feature). The dataset consists of Android methods annotated with their corresponding category. We use the same set of labels as SuSi for easy comparison, which includes: Log, NFC, Audio, User Account, Text Message, File System, Network connections, System Settings, Geolocation, Bluetooth, Contacts, and Database. For the Pair-encoder, we use the pre-trained model `bart-large` [7] fine-tuned with the multi_nli dataset [8] (a crowd-sourced collection of around 433k sentence pairs annotated with textual entailment information). The document formats used for this experiment are described in Table 6.3.

Table 6.9 shows the precision and recall of the classifiers. Format E achieves the best overall performance using the method and class description. Using only the class description (Format F) still gets an equivalent precision compared to SuSi and improves the recall by 10%. By manually analysing the predictions per category, we observe that SuSi fails to assign a semantic category (no-category label) to a large percentage of methods. This performance contrasts the SuSi source and sinks classifiers, where the results are comparable to `DocFlow` under many settings. One reason for this might be that extracting features from the code does not lead to good models to predict semantic categories. In contrast, our Semantic Category classifier uses natural language

[7] huggingface.co/facebook/bart-large
[8] huggingface.co/datasets/multi_nli

descriptions, which are a more natural choice for this task. Another disadvantage of SuSi is that its feature extraction needs to be modified to add a new category. `DocFlow` zero-shot approach does not require any internal change to add new labels.

	Accuracy	**Precision**	**Recall**	**F1-score**
`DocFlow` (E)	**0.86**	**0.91**	**0.86**	**0.88**
`DocFlow` (F)	0.83	0.89	0.83	0.86
`DocFlow` (G)	0.79	0.89	0.70	0.78
SuSi	0.59	0.88	0.60	0.71

Table 6.9: Docflow and SuSi semantic category classification results. Precision and recall metrics are calculated for each label, and their weighted average is displayed. Formats used for `DocFlow` runs are shown within parentheses

6.4 Discussion

Our experiments focus mainly on detecting sources and sinks. Security specifications can include other method types. Piskachev *et al.* [61] use a broader definition of *Security-Relevant Methods* to include methods types such as sanitisers and authentication. They modified SuSi to include more general features that can be used for other Java-based projects. `DocFlow` can be adapted to detect other method types by providing additional training data with more labels. Furthermore, we note that `DocFlow` is more precise at detecting sources than sinks. The main reason is because the FP rate for sinks is higher than sources. This problem can be addressed by providing more granular labels. For instance, we have shown (in Section 6.3.5) that `DocFlow` detects callbacks with high precision.

`DocFlow` is flexible regarding the embedding and algorithm choices. We tested `DocFlow` with other well-known embeddings such as Word2Vec [100] and Universal Sentence Encoder [189], but the overall results were better with Sentence-BERT. Instead of a transfer learning approach, other works [95, 101, 108] use a fine-tuning approach for the classification task. This approach adds an output layer after calculating the embeddings and retrains the whole network, optionally fixing the weights of one or more layers. Peters *et al.* showed that both approaches produce similar results in most cases [190]. We chose a transfer learning approach as it enables us to easily reuse the embeddings for other tasks, such as semantic search and clustering. Still, `DocFlow`'s classifier can be used with a fine-tuning classification approach.

Limitations. `DocFlow` assumes that Android APIs are well documented in terms of completeness and quality. We have validated the completeness of the documentation (Android framework and Google libraries), but evaluating its quality is more complex. We rely on the fact that Android is a popular platform with millions of developers worldwide, and this forces framework developers to add good documentation following a regular coding and documentation style. Because of this, `DocFlow` is not suited for all types of projects and focuses on well-documented projects.

Recently, Liu *et al.* studied the problem of silently evolved methods in Android [180]. These are methods that receive code updates from developers but their documentation is not adjusted.

This scenario can introduce noise in our framework. In the case of Android, these are small changes in the implementation but are unlikely to change the method's purpose or semantics, as the API methods need to be compatible with previous versions. Still, we have shown in Section 6.3.4 that `DocFlow` adapts well to small changes in the documentation.

`DocFlow`'s crawler could fail to parse the documentation properly. This is due to complex cases that our crawler could handle better. For instance, an HTML page might contain rare annotations that break the parsing. We fixed many of these while testing, but there could be some missing cases given the volume of pages. Special tokens add complexity to the parsing, e.g., acronyms and constants. Consider the following tokens that represent an SSL handshake: `SSL_HANDSHAKE`, `SSL-HANDSHAK`, `SSLHandshake`, and `sslHandshake`. Failing to parse these tokens properly can generate noise and reduce the quality of the embeddings. We use an iterative approach to improve `DocFlow`'s parser.

`DocFlow` ignores deprecated methods by defaults as these might show only a deprecated message. An alternative approach can either download a previous version (where the method is not deprecated) or use the description of the new method that replaces the deprecated if there is a reference to it in the documentation.

We have shown that `DocFlow` achieves good performance across time by analysing different versions of the Location package. However, this might not be true for other framework packages or Google libraries. This limitation is associated with the complexity of manual validation, as we need to produce labels for each package. Moreover, the labelling process might also add errors to the evaluation. We address this problem by cross-labelling the dataset with three experts. We discussed an alternative approach in the next section.

6.5 Related Work

`DocFlow` is based on the principle of *Dual-Channel Software Security* [92]. In this approach, software is studied from a dual channel perspective, Natural Language Processing and Software Engineering. Recent advances in this area have introduced novel control and data-flow analyses guided by identifiers from the program [191, 192] and subsequently applied this to software testing [193]. We apply NLP to seed taint analysis by auto-identification of sources and sinks from software documentation. NLP techniques have been widely used to study apps issues such as privacy policies [108–111], permissions [106, 107], and app descriptions [104, 105].

Several works use embeddings to represent program code and reason about the semantics [194–198]. For instance, Ding *et al.* used Word embedding techniques to identify physical channels in IoT from application descriptions [118]. Tian *et al.* use Dual-Channel Security to automatically collect security-relevant information from an IoT app's description and code, represent with Word2Vec and to compare them with actual behaviour using program analysis [199]. *Li et al.* propose Word2API, a model that relates comments to API code to solve the semantic gap between both [200]. Alternatively, `DocFlow` can use a similar approach to generate the embeddings. However, this requires parsing the Android framework source code and will fail for

Google proprietary libraries.

The problem of detecting sensitive methods has been studied using Machine Learning [51,61] or other statistical approaches [62]. One limitation of these works is that they rely on code analysis for feature extraction, but this is not always possible (e.g., Google Play Services libraries). For instance, SuSi extracts 144 features from each application before feeding the feature vectors to an SVM classifier. Obtaining the Android framework code is not straightforward as the Android jar file is shipped with stub methods, and researchers need to develop custom scripts to extract the code for different versions due to constant changes in the Android architecture. In contrast, our approach automatically extracts vector representations using sentence embeddings from Android API documentation, which is publicly available.

A different approach proposes a query specification language for taint-flows definition [63]. This approach requires input from developers to specify taint-flow queries. `DocFlow` takes a different approach and analyse software documentation instead of relying on developers input. Seldon is a semi-supervised method for inferring taint analysis specifications [64], and USpec [65] is an unsupervised tool for discovering aliasing specifications of APIs. Both tools learn from a large dataset of programs. We focus on API documentation instead of programs.

Collaborative Labelling. Data labelling is a common practice for learning-based experiments. Labelling techniques include annotation from experts, extracting relations from available sources, automatic label generation, and crowdsourcing. `DocFlow` (and the baselines) require a dataset with annotated labels for training. Moreover, we classified methods from the entire Android framework and Google libraries, but the results were validated for a small number of annotated classes and methods. For this, we rely on expert labelling following an iterative approach. The problem with this approach is that it takes time to scale for larger datasets. For example, it took eight years to produce the Pen Treebank corpus for POS tagging in NLP using linguistic experts [201].

Given the scalability issues, another suitable alternative to construct datasets is crowdsourcing. This technique aims to create labelled datasets at a lower cost by using crowdworkers for the labelling task while the experts define the procedure [202]. A crowdsourcing approach would enable an efficient way to construct a large dataset of sources and sinks. Similar projects have been developed in the past [4, 203, 204]. Despite its benefits, a crowdsourcing approach has limitations; It is still challenging to obtain high-quality labels because of poor guidelines, poor work from crowdworkers, confusing interfaces, prior knowledge, and task complexity. Chang *et al.* give an overview of the techniques to improve the quality of crowdsourced labels and propose a framework for collaborative crowdsourcing for machine learning dataset [205]. Relying on the software community for the labelling task can address some of these issues.

6.6 Chapter Summary

This chapter explores the means of analysing arbitrary platforms efficiently RQ3. Our study of the Wear and Android TV platform allows us to identify one task common to all platforms:

generating taint specifications. This task is time-consuming and prone to errors. We identify that current automated approaches rely on platform source code, which is not always available. In contrast, we argue that Android documentation, publicly available and easy to access, contains rich semantic information that can be used to generate taint specifications for arbitrary platforms. To implement this idea, we have developed `DocFlow`.

`DocFlow` is a Natural Language Processing framework for analysing Android methods. `DocFlow` can detect sources and sinks, classify methods according to their semantic category or identify clusters of similar methods. Our novel approach removes the need to analyse code, which is particularly convenient when source code is not available or complete. Thus, `DocFlow` enables the security community to incorporate popular closed-source APIs into taint specifications.

We evaluate `DocFlow` on a corpus from the Android AOSP and Google Play documentation with over 45000 methods. Our results show that `DocFlow` identifies sensitive methods and their semantic categories with better precision than previous tools. `DocFlow` can also be used to find semantically similar methods across versions of the same platform, helping analysts to cope with the constant updates of the Android framework and Google Play Services libraries.

Our framework will help the community to obtain better taint specifications, resulting in a more robust security information flow analysis of Android applications across platforms RQ1. These capabilities open a new perspective for other platforms where developers heavily rely on closed-source API methods and facilitate the analysis of new platforms or releases.

7 Conclusions

We conclude this thesis by summarising our contributions and revisiting the research questions by reflecting on our findings and how they portray in the Android ecosystem. Finally, we discuss potential future work following our line of research.

7.1 Thesis Summary

At the beginning of this thesis, we specified the objective of this work: *We aim to improve the security and privacy of Android apps across different platforms.* To achieve this, we study multi-platform apps and how information flows differ across platforms. In particular, we consider the Wear OS and Android TV platforms to approximate the behaviour of apps on arbitrary platforms.

Our first step consisted of a study on the composition of these platforms that shows structural differences in apps from different platforms, presented in Chapter 3. This chapter also introduces the Google Play Services libraries and describes how its architecture enables new communication channels. One such channel is the Data Layer, which exposes APIs to transmit data across mobile and wearable apps.

The Data Layer enables the transmission of sensitive data across devices in the mobile-wear ecosystem. This channel could provoke involuntary or intentional data leaks. In Chapter 4, we study the problem of detecting sensitive information flows that use this communication channel. Our solution, `WearFlow`, is a static analysis framework that augments taint analysis capabilities and accurately models mobile-wear communications. `WearFlow` achieves better precision than the baseline tool and scales to detect sensitive flows in real-world apps. Our contribution enables the detection of information flows that otherwise would remain undetected.

Applications in Android TV differ substantially from Wear OS apps in terms of purpose, data they handle, and interactions. Therefore, Chapter 5 presents a deep analysis of the Android TV ecosystem, which includes information flows, third-party libraries, communication APIs and permissions. We compared TV apps to their mobile counterpart using static analyses and confirmed many findings using network traffic analysis and manual verification. The results of this evaluation show pervasive sensitive data collection, including the collection of static identifiers and viewing history. Most of these behaviours can be attributed to tracking and advertising services. On top of this, we found bad development practices by looking at permissions and API calls. These problems leave Android TV users vulnerable to attacks and data leakage. We identify policies that can result in a safer ecosystem, such as better documentation and guidelines to develop TV apps and a privacy-aware Play Store interface for Android TV.

The last contribution of this thesis centres around automating the analysis of arbitrary app platforms. While a fully automated approach is unfeasible, we partially address this issue by generating taint specifications that can be used as input to study information flows in apps from arbitrary platforms. In Chapter 6, we propose `DocFlow`. This end-to-end framework generates taint specifications for the Android framework and Google libraries using a novel approach that does not require access to source code. `DocFlow` uses software documentation and Natural Language Processing to detect security-sensitive methods. Our framework achieves better performance than baseline tools and is robust to software evolution and changes in the documentation.

Based on the summary presented above, we now revisit the three research questions that we have defined in the introduction of this thesis.

RQ1. Can we detect sensitive information flows in arbitrary app platforms?

The characterisation presented in Chapter 3 allowed us to identify several complex abstractions and modes of interaction that are not being considered by state-of-the-art frameworks. The complexity of the Data Layer shows that, in some cases, static taint tracking requires custom models to propagate sensitive information flow with precision. Our results from Chapter 4 reveal that the precision of the baseline tool in the mobile platform does not replicate automatically in the wearable platform. Similarly, a dynamic analysis approach also requires reversing the communication protocols and data exchange mechanism. While `WearFlow` is simpler and does not require extra hardware, both solutions require custom analyses to capture the semantics of available APIs and interactions. We acknowledge that other platforms and libraries might use different APIs that result in new modes of interaction not covered in this thesis. These behaviours must be soundly modelled to avoid overestimating information flows to impractical levels. This leaves the road open to future work discussed in the section below. Considering our results, we conclude that detecting information flows in arbitrary Android platforms is possible, but this requires extra effort to model platform-specific abstractions. We hope our methodology can be replicated in other works to cover abstractions we did not consider.

RQ2. Do other platforms present problems unseen in the mobile platform?

Analysing wearable and TV apps revealed several differences with the mobile platform. The preliminary comparison of TV, wearable and mobile apps offered the first insight in Chapter 3. The results show structural differences across platforms (permissions, libraries, APIs). The Data Layer is a clear example, as this API is not available for TV apps and are irrelevant when looking at mobile apps in isolation. Our work enables the detection of this type of information flow that otherwise would remain undetected. Considering the study of TV apps in Chapter 5, we detected that app developers use permissions inconsistently. This problem does not manifest for mobile apps, as the problem originates due to poor guidelines for porting mobile apps to Android TV. The considerably lower user rating of TV apps compared to their mobile counterpart suggests that developers give less attention to this platform. Our evaluation of the TV and mobile apps showed differences in data collection practices, inter-device communication, and the limitation

of using repositories of third-party libraries collected from mobile apps in TV apps.

All these results are evidence of platform-specific problems and the need to give more attention to apps across different platforms. It is important to mention that the last years have seen an increase in scientific publications for less popular platforms such as Android TV and Wear OS compared to the beginning of this research. This thesis contributes to a better understanding of arbitrary app ecosystems. Still, a collective effort is required from Android developers, app developers, and researchers to improve the user experience and security in these ecosystems.

RQ3. Is it possible to automate the analysis of information flows in arbitrary platforms?

To answer this question, we need to portray our findings in the Android ecosystem and consider the practicalities of analysing arbitrary platforms. The answer to RQ1 already provides a glimpse into this question. Different abstractions and modes of interaction prevent using off-the-shelf tools on arbitrary platforms. A precise analysis requires modelling platform-specific behaviour, e.g., APIs and third-party libraries. However, our experience analysing three app ecosystems (mobile, wearable, and TV) taught us that common tasks could be automated. One such task is generating taint specifications. Producing a list of sources and sinks for each platform is time-consuming and prone to errors. This requires reading API documentation, development guides, and potentially the source code. In this context, we developed an end-to-end framework that uses NLP to classify sensitive methods. Our solution is robust against software evolution, does not need access to source code and performs better than baseline tools. Thus, `DocFlow` is an excellent option to automate the generation of taint specifications. While `DocFlow` solve only one part of the problem, it is an important step to automate information flow analysis for arbitrary platforms in the Android ecosystem.

7.2 Future Work

This research leaves the way open to follow-up works in many directions. We formulate some of these directions below.

General Framework. We have shown that `DocFlow` is an efficient framework for classifying sensitive methods. However, its output needs to be used by a static analyser to be practical. For instance, `DocFlow` can be integrated with `WearFlow` to create a framework where `DocFlow` is the first step of a pipeline that analyses information flows. Moreover, `DocFlow` can be integrated with any framework that receives taint specifications as input, such as FlowDroid. Currently, this integration requires custom development. Future work could integrate all the frameworks into one cohesive unit suitable for analysing information flows in multi-platform apps.

Other Platforms and interactions. We have presented a study of multi-platform apps and their interaction. We have covered two platforms, but more are available in the Android ecosystem and from other vendors. For instance, Android offers two variants for its vehicle platform: `Android Auto`, an interface that provides a driver-optimised experience for smartphone users

and `Android Automotive OS`, a fully customised Android built into vehicles. These variants present different modes of interaction and can be integrated with Google Assistant. We have laid the foundations to study arbitrary platforms. Future research efforts can use our methodology to analyse new platforms and interactions efficiently.

Third-party libraries analysis. We have analysed documentation from Android APIs and Google libraries. However, app developers also use third-party libraries that offer rich documentation. Embedded libraries are usually bigger than app components and inherit permissions from the host app. Moreover, popular libraries must produce good documentation to support their users, in the same way as Android framework developers. A potential future work can involve understanding library APIs by analysing their documentation. Static and dynamic analyses can leverage this information to enhance the understanding of third-party code behaviour.

User perception. In this thesis, we have studied information flows on multiple platforms and across devices by detecting data leaks or vulnerabilities. However, it is also important to understand the users' perceptions of privacy in interconnected ecosystems. We have shown how inter-device channels require custom analysis due to their complexity. This complexity also affects final users who need to understand how data is shared between devices when paired or otherwise. A user study on privacy in interconnected settings can shed more light and contribute to better guidelines and user interfaces that inform users about the information they share, resulting in more secure ecosystems.

Appendix A: WearBench Apps Description

This section offers further details about each test case from `WearBench`. All apps were developed using Java, and we implemented all APIs from the Data Layer. We focused on scenarios where sensitive and non-sensitive data is transmitted using Data Layer APIs. We describe each test case grouped by API type below.

Message Client API

This API allows apps to send asynchronous messages and attach the following items: 1) arbitrary payload 2) a path that uniquely identifies the message's action. Messages are a one-way communication mechanism that can be used for sharing short messages or remote procedure calls, such as starting an activity on other devices.

The following code snippets show a MessageClient APIs sending (Listings 1) and receiving (2) asynchronous messages. We show a simple example and then describe the different test cases.

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    MessageClient messageClient = Wearable.getMessageClient(this);
    String deviceId = source();
    String path ="/path"
    String nodeId = getNodeID();
    messageClient.sendMessage(nodeId, path, deviceId.getBytes());
    messageClient.sendMessage(nodeId, "path_empty", "empty".getBytes());
}
```

Listing 1: MessageClient - send message

```
public class MyService extends WearableListenerService {
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
            if(messageEvent.getPath().equals("/path")){
                String message = new String(messageEvent.getData());
                Log.i("Leak", message);
            } else if (messageEvent.getPath().equals("/path_empty")){
                String text = new String(messageEvent.getData());
                Log.i("Empty", text);
            }
    }
}
```

Listing 2: MessageClient - receive message

- MessageClient1: Mobile app sends a single tainted value using a MessageClient. The wearable app receives the tainted data on a MessageEvent. **Challenge**: In the mobile app, the path value has to be calculated based on the context. In the wearable, the path

has to be fetched from the Manifest and instrumented in the WearableListenerService.

- SimpleMessage2: Similar to MessageClient1. This example also sends a constant string through a second channel. The wearable app receives both values. The path for the Service is declared in the Manifest. **Challenge**: The analysis must distinguish sensitive and non-sensitive channels.

- SimpleMessage3: Similar to MessageClient1. This example does not leak any sensitive data. The wearable app expects an incorrect path. **Challenge**: The analysis has to identify unreachable code in the wearable app and report no leaks.

- SimpleMessage4: Similar to MessageClient1, but uses the old deprecated wearable API. In this test case, the context of the transmission is spread across different lifecycle methods. **Challenge**: Modelling of wearable lifecycle methods, context extraction, and sensitive channel detection.

DataClient API

A DataClient API uses DataItem abstractions to synchronise data between smartphones and smartwatches. A DataItem is composed of a path and payload. Like the MessageClient API, the path works as an event identifier. The payload consists of a byte array that stores the data (up to 100 KB). Typically, the payload is further encapsulated on a DataMap object. At the same time, the DataMap is encapsulated into a request object. The following code snippets illustrates each step, the synchronisation event in Listing 3 and deliver event in Listing 4. Note that this is only one way to trigger the synchronisation process and we have implemented all of them across the test cases.

```
1  public class MainActivity extends Activity {
2      private static final String COUNT_KEY = "com.example.key.count";
3      private DataClient dataClient;
4      private int count = 0;
5      ...
6      // Create a data map and put data in it
7      private void increaseCounter() {
8          PutDataMapRequest putDataMapReq =
               PutDataMapRequest.create("/count");
9          putDataMapReq.getDataMap().putInt(COUNT_KEY, count++);
10         PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
11         Task<DataItem> putDataTask = dataClient.putDataItem(putDataReq);
12     }
13   ...
14 }
```

Listing 3: DataClient - The DataItem is synchronised with the putDataItem call

```
1  public class MainActivity extends Activity implements
       DataClient.OnDataChangedListener {
2      private static final String COUNT_KEY = "com.example.key.count";
3      private int count = 0;
4
5      @Override
6      public void onDataChanged(DataEventBuffer dataEvents) {
7          for (DataEvent event : dataEvents) {
8              if (event.getType() == DataEvent.TYPE_CHANGED) {
9                  // DataItem changed
10                 DataItem item = event.getDataItem();
11                 if (item.getUri().getPath().compareTo("/count") == 0) {
12                     DataMap dataMap =
                           DataMapItem.fromDataItem(item).getDataMap();
13                     updateCount(dataMap.getInt(COUNT_KEY));
14                 }
15             } else if (event.getType() == DataEvent.TYPE_DELETED) {
16                 // DataItem deleted
17             }
18         }
19     }
20 }
```

Listing 4: DataClient - receives synchronisation event

- SimpleDataItem1: The mobile app adds a tainted value and a constant number to a DataItem. The wearable app receives the event on a service. Challenge: The analysis has to differentiate between the tainted variable and the constant number. The keys for DataMaps values have to be computed. The channel path has to be calculated from the Manifest.

- SimpleDataItem2: In this test case, two sensitive variables are synchronised using DataMaps. Challenge: The request encapsulates the DataMap using a different API.

- SimpleDataItem3: In this test case, two synchronisation events are triggered in the mobile app. Challenge: Assign the data flows to the corresponding event.

- SimpleDataItem4: In this test case, the synchronisation events are triggered in a lifecycle method using the APIs for Wear OS 1. Challenge: Model the previous version of the Data Layer APIs and lifecycle methods.

- InterDataItem1: In this test case, the context of the communication is created across different methods. The code uses a different API to construct the DataMap. Challenge: The analysis has to do an inter-procedural analysis to identify the context of the communication.

- InterDataItem2: This test case is similar to InterDataItem1, but the communication con-

text is more complex. Challenge: Same as InterDataItem1 but more complex.

- Asset: The wearable app sends 2 Assets with tainted values using 2 different channels. Challenge: The wearable app uses 2 different APIs to encapsulate the Assets. The analysis has to match the data flows to the correct channel.

ChannelClient API

The ChannelClient can transfer data between mobile and wearable apps. This API is designed to handle large files and streaming, e.g., voice data. The Listings 5 and 6 show this interaction.

```
1  public class MainActivityMobile extends Activity  {
2
3      private ChannelClient channelClient;
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8          channelClient = Wearable.getChannelClient(this);
9          File file = getTaintedFile(deviceID);
10         Task<ChannelClient.Channel> channelTask =
               channelClient.openChannel("*","/my_path");
11         ChannelClient.Channel channel = channelTask.getResult();
12         channelClient.sendFile(channel, Uri.fromFile(file));
13     }
14 }
```

Listing 5: ChannelClient - The file is sent with the sendFile call

```
1  public class ChannelListenerService extends WearableListenerService {
2      @Override
3      public void onChannelOpened(ChannelClient.Channel channel) {
4          String pathName = "/sdcard/file.txt";
5          if (channel.getPath().equals("/my_path")) {
6              File file = getFile(pathName);
7              Uri uri = Uri.fromFile(file);
8              Wearable.getChannelClient(this).receiveFile(channel,uri,true);
9              byte[] data = readBytesFromUri(uri);
10         }
11     }
12 }
```

Listing 6: DataClient - The file is received in the callback

- SimpleChannel: The Mobile app adds a file into a channel and sends it to the wearable app. The wearable app receives the file on the channel and reads the tainted value from the file. Challenge: Model the interaction of the ChannelClient APIs.

Bibliography

[1] P. Associates, "Future of video." Accessed March 2021. http://www.parksassociates.com/events/future-of-video/fov-2018-pr5.

[2] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, "All things considered: an analysis of iot devices on home networks," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1169–1185, 2019.

[3] V. Sivaraman, H. H. Gharakheili, C. Fernandes, N. Clark, and T. Karliychuk, "Smart iot devices in the home: Security and privacy implications," *IEEE Technology and Society Magazine*, vol. 37, no. 2, pp. 71–79, 2018.

[4] D. Y. Huang, N. Apthorpe, F. Li, G. Acar, and N. Feamster, "Iot inspector: Crowdsourcing labeled network traffic from smart home devices at scale," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 4, no. 2, pp. 1–21, 2020.

[5] D. P. Release, "Consumers benefit from virtual experiences, but need help managing screen time, security and tech overload." Accessed August 2022. https://www2.deloitte.com/us/en/pages/about-deloitte/articles/press-releases/connectivity-and-mobile-trends.html/.

[6] H. Fortify, "Internet of things security study: smartwatches." Accessed March 2020, 2015. https://www.ftc.gov/system/files/documentspublic_comments/2015/10/00050-98093.pdf.

[7] H. Wang, T. T.-T. Lai, and R. Roy Choudhury, "Mole: Motion leaks through smartwatch sensors," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pp. 155–166, 2015.

[8] R. Goyal, N. Dragoni, and A. Spognardi, "Mind the tracker you wear: a security analysis of wearable health trackers," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp. 131–136, 2016.

[9] Q. Do, B. Martini, and K.-K. R. Choo, "Is the data on your wearable device secure? an android wear smartwatch case study," *Software: Practice and Experience*, vol. 47, no. 3, pp. 391–403, 2017.

[10] J. Blasco, T. Chen, H. Kupwade Patil, and D. Wolff, *Wearables Security and Privacy*, pp. 351–380. Studies in Systems, Decision and Control, Switzerland: Springer International Publishing AG, Jan. 2019.

[11] J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne, "Characterization of early smartwatch apps," in *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pp. 1–6, IEEE, 2016.

[12] L. Cilliers, "Wearable devices in healthcare: Privacy and information security issues," *Health information management journal*, vol. 49, no. 2-3, pp. 150–156, 2020.

[13] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi, "Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach," in *Proceedings of the Internet Measurement Conference*, pp. 267–279, 2019.

[14] A. Nikas, E. Alepis, and C. Patsakis, "I know what you streamed last night: On the security and privacy of streaming," *Digital Investigation*, vol. 25, pp. 78–89, 2018.

[15] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *28th {USENIX} Security Symposium*, pp. 1133–1150, 2019.

[16] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive information tracking in commodity iot," in *27th {USENIX} Security Symposium*, pp. 1687–1704, 2018.

[17] R. Binns, U. Lyngs, M. Van Kleek, J. Zhao, T. Libert, and N. Shadbolt, "Third party tracking in the mobile ecosystem," in *Proceedings of the 10th ACM Conference on Web Science*, pp. 23–31, 2018.

[18] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," in *Network and Distributed System Security Symposium*, 2018.

[19] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 356–367, 2016.

[20] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th international conference on software engineering companion*, pp. 653–656, 2016.

[21] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 141–152, IEEE, 2018.

[22] L. Li, T. F. Bissyandé, J. Klein, and Y. Le Traon, "An investigation into the use of common libraries in android apps," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 403–414, IEEE, 2016.

[23] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond google play: A large-scale comparative study of chinese android app markets," in *Proceedings of the Internet Measurement Conference 2018*, pp. 293–307, 2018.

[24] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl, "Block me if you can: A large-scale study of tracker-blocking tools," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 319–333, IEEE, 2017.

[25] F. T. Commission, "Vizio to pay $2.2 million to ftc." Accessed March 2021. `https://www.ftc.gov/news-events/press-releases/2017/02/vizio-pay-22-million-ftc-state-new-jersey-settle-charges-it`.

[26] WIRED, "Millions of google, roku, and sonos devices are vulnerable to a web attack — wired.." Accessed March 2021. `https://www.wired.com/story/chromecast-roku-sonos-dns-rebinding-vulnerability/.`.

[27] G. Acar, D. Y. Huang, F. Li, A. Narayanan, and N. Feamster, "Web-based attacks to discover and control local iot devices," in *Proceedings of the 2018 Workshop on IoT Security and Privacy*, pp. 29–35, 2018.

[28] H. Mohajeri Moghaddam, G. Acar, B. Burgess, A. Mathur, D. Y. Huang, N. Feamster, E. W. Felten, P. Mittal, and A. Narayanan, "Watching you watch: The tracking ecosystem of over-the-top tv streaming devices," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 131–147, 2019.

[29] J. Varmarken, H. Le, A. Shuba, A. Markopoulou, and Z. Shafiq, "The tv is smart and full of trackers: Measuring smart tv advertising and tracking," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 129–154, 2020.

[30] N. Malkin, J. Bernd, M. Johnson, and S. Egelman, ""what can't data be used for?" privacy expectations about smart tvs in the us," in *Proceedings of the 3rd European Workshop on Usable Security (EuroUSEC), London, UK*, 2018.

[31] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2016.

[32] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[33] F. Wei, S. Roy, X. Ou, *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1329–1341, ACM, 2014.

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

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

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

[37] X. Cui, J. Wang, L. C. Hui, Z. Xie, T. Zeng, and S.-M. Yiu, "Wechecker: efficient and precise detection of privilege escalation vulnerabilities in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 1–12, 2015.

[38] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "Jucify: a step towards android code unification for enhanced static analysis," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1232–1244, 2022.

[39] A. Bosu, F. Liu, D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 71–85, 2017.

[40] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 318–329, 2016.

[41] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1137–1150, 2018.

[42] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 421–431, 2018.

[43] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: Compositional analysis of android inter-app permission leakage," *IEEE transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[44] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 543–558, 2013.

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

[46] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[47] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 603–620, 2019.

[48] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones.," in *NDSS*, vol. 11, pp. 17–33, 2011.

[49] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," *EuroSec, April*, 2013.

[50] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning–based text classification: a comprehensive review," *ACM computing surveys (CSUR)*, vol. 54, no. 3, pp. 1–40, 2021.

[51] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," *University of Darmstadt, Tech. Rep. TUDCS-2013*, vol. 114, p. 108, 2013.

[52] H. Wang, H. Li, and Y. Guo, "Understanding the evolution of mobile app ecosystems: A longitudinal measurement study of google play," in *The World Wide Web Conference*, pp. 1988–1999, 2019.

[53] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of api evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[54] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Nearby threats: Reversing, analyzing, and attacking google's' nearby connections' on android," in *Network and Distributed System Security Symposium*, 2019.

[55] Google, "Google play store." Accessed March 2021. `https://play.google.com/`.

[56] Google, "Android official documentation." Accessed January 2022. `https://developer.android.com/guide/components/activities/activity-lifecycle`.

[57] T. Verge, "There are over 3 billion active android devices." Accessed January 2023. `https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021/`.

[58] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 3, pp. 1–35, 2021.

[59] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *2013 IEEE International Conference on Software Maintenance*, pp. 70–79, 2013.

[60] P. Calciati, K. Kuznetsov, X. Bai, and A. Gorla, "What did really change with the new release of the app?," in *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 142–152, 2018.

[61] G. Piskachev, L. N. Q. Do, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 181–191, 2019.

[62] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," *SIGPLAN Not.*, vol. 44, p. 75–86, jun 2009.

[63] G. Piskachev, J. Späth, I. Budde, and E. Bodden, "Fluently specifying taint-flow queries with fluenttql," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–33, 2022.

[64] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev, "Scalable taint specification inference with big code," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 760–774, Association for Computing Machinery, 2019.

[65] J. Eberhardt, S. Steffen, V. Raychev, and M. Vechev, "Unsupervised learning of api aliasing specifications," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), p. 745–759, Association for Computing Machinery, 2019.

[66] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild,"

in *International conference on security and privacy in communication systems*, pp. 172–192, Springer, 2018.

[67] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the seventh european workshop on system security*, pp. 1–6, 2014.

[68] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *Proceedings of the 26th International Conference on World Wide Web*, pp. 143–152, 2017.

[69] Z. Xu, C. Shi, C. C.-C. Cheng, N. Z. Gong, and Y. Guan, "A dynamic taint analysis tool for android app forensics," in *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 160–169, IEEE, 2018.

[70] IBM, "Hlc app scan." Accessed October 2022. `https://www.hcltechsw.com/appscan/offerings/standard`.

[71] CyberRes, "Fortify static code analyzer." Accessed October 2022. `https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer`.

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

[73] Y.-C. Lin, "Androbugs framework: An android application security vulnerability scanner," *Blackhat Europe*, vol. 2015, 2015.

[74] A. Avancini and M. Ceccato, "Security testing of the communication among android applications," in *2013 8th International Workshop on Automation of Software Test (AST)*, pp. 57–63, IEEE, 2013.

[75] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[76] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

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

[78] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 331–341, 2018.

[79] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 176–186, 2018.

[80] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, pp. 214–224, IBM Corp., 2010.

[81] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis, "Static analysis of java enterprise applications: frameworks and caches, the elephants in the room," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 794–807, 2020.

[82] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the ifds algorithm," in *International Conference on Compiler Construction*, pp. 124–144, Springer, 2010.

[83] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[84] D. Anthony and G. Geoffroy, "Androguard documentation." Accessed February 2023. `https://github.com/androguard/androguard`.

[85] Y. Z. X. Jiang, "Detecting passive content leaks and pollution in android applications," in *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[86] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing android," *IEEE Transactions on Software Engineering*, vol. 40, no. 6, pp. 617–632, 2014.

[87] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[88] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications.," in *NDSS*, vol. 14, pp. 23–26, 2014.

[89] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp. 37–48, 2015.

[90] C. Rizzo, L. Cavallaro, and J. Kinder, "Babelview: Evaluating the impact of code injection attacks in mobile webviews," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 25–46, Springer, 2018.

[91] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and scalably vetting javascript bridge in android hybrid apps," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 143–166, Springer, 2017.

[92] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '20, (New York, NY, USA), p. 25–28, Association for Computing Machinery, 2020.

[93] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 5–14, IEEE, 2013.

[94] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[95] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?," in *China national conference on Chinese computational linguistics*, pp. 194–206, Springer, 2019.

[96] B. Srinivasa-Desikan, *Natural Language Processing and Computational Linguistics: A practical guide to text analysis with Python, Gensim, spaCy, and Keras*. Packt Publishing Ltd, 2018.

[97] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[98] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[99] G. E. Hinton, "Distributed representations," tech. rep., Carnegie Mellon University, 1984.

[100] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[101] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[102] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[103] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[104] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th international conference on software engineering*, pp. 1025–1035, 2014.

[105] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1354–1365, 2014.

[106] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "{WHYPER}: Towards automating risk assessment of mobile applications," in *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 527–542, 2013.

[107] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A large-scale empirical study on android runtime-permission rationale messages," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 137–146, IEEE, 2018.

[108] P. Nema, P. Anthonysamy, N. Taft, and S. T. Peddinti, "Analyzing user perspectives on mobile app privacy at scale," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 112–124, 2022.

[109] H. Harkous, K. Fawaz, R. Lebret, F. Schaub, K. G. Shin, and K. Aberer, "Polisis: Automated analysis and presentation of privacy policies using deep learning," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 531–548, 2018.

[110] S. Zimmeck, P. Story, D. Smullen, A. Ravichander, Z. Wang, J. R. Reidenberg, N. C. Russell, and N. Sadeh, "Maps: Scaling privacy compliance analysis to a million apps," *Proc. Priv. Enhancing Tech.*, vol. 2019, p. 66, 2019.

[111] N. Mehdy, C. Kennington, and H. Mehrpouyan, "Privacy disclosures detection in natural-language text through linguistically-motivated artificial neural networks," in *International Conference on Security and Privacy in New Computing Environments*, pp. 152–177, Springer, 2019.

[112] R. Witte, Q. Li, Y. Zhang, and J. Rilling, "Text mining and software engineering: an integrated source code and document analysis approach," *Iet Software*, vol. 2, no. 1, pp. 3–16, 2008.

[113] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "Dase: Document-assisted symbolic execution for improving automated software testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 620–631, IEEE, 2015.

[114] Y. Sun, A. K. Wong, and M. S. Kamel, "Classification of imbalanced data: A review," *International journal of pattern recognition and artificial intelligence*, vol. 23, no. 04, pp. 687–719, 2009.

[115] Y. Aafer, W. You, Y. Sun, Y. Shi, X. Zhang, and H. Yin, "Android smarttvs vulnerability discovery via log-guided fuzzing," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[116] G. V. Research, "Smart home market size, share and trends analysis report by application." Accessed November 2022, 11 2022. `https://www.grandviewresearch.com/industry-analysis/smart-homes-industry`.

[117] Google, "Android blog." Accessed January 2022. `https://blog.google/products/android/io22-multideviceworld/`.

[118] W. Ding and H. Hu, "On the safety of iot device physical interaction control," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 832–846, 2018.

[119] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, 2019.

[120] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot.," in *NDSS*, 2019.

[121] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," *arXiv preprint arXiv:1905.02713*, 2019.

[122] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 226–237, 2016.

[123] Google, "Google play services documentation." Accessed January 2023. `https://developers.google.com/android/guides/setup`.

[124] Google, "Android blog support." Accessed January 2022. `https://developer.android.com/training/wearables`.

[125] Google, "Wearable api documentation." Accessed January 2023. `https://developers.google.com/android/reference/com/google/android/gms/wearable/DataMap`.

[126] Google, "Android tv guide." Accessed March 2021. `https://developer.android.com/training/tv/start/start`.

[127] W. Bu, M. Xue, L. Xu, Y. Zhou, Z. Tang, and T. Xie, "When program analysis meets mobile security: an industrial study of misusing android internet sockets," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 842–847, 2017.

[128] D. Wu, D. Gao, R. K. Chang, E. He, E. K. Cheng, and R. H. Deng, "Understanding open ports in android applications: Discovery, diagnosis, and security assessment," in *Network and Distributed System Security Symposium*, Internet Society, 2019.

[129] Y. J. Jia, Q. A. Chen, Y. Lin, C. Kong, and Z. M. Mao, "Open doors for bob and mallory: Open port usage in android apps and security implications," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 190–203, IEEE, 2017.

[130] Google, "Nearby connections guide." Accessed March 2021. `https:https://developers.google.com/nearby/connections/android/manage-connections#authenticate_a_connection`.

[131] Y. Liu, X. Chen, Y. Liu, P. Kong, T. F. Bissyande, J. Klein, X. Sun, C. Chen, and J. Grundy, "A comparative study of smartphone and smart tv apps," *arXiv preprint arXiv:2211.01752*, 2022.

[132] A. K. Sikder, G. Petracca, H. Aksu, T. Jaeger, and A. S. Uluagac, "A survey on sensor-based threats and attacks to smart devices and applications," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1125–1159, 2021.

[133] K. W. Ching and M. M. Singh, "Wearable technology devices security and privacy vulnerability analysis," *International Journal of Network Security & Its Applications*, vol. 8, no. 3, pp. 19–30, 2016.

[134] D. Li, Y. Lyu, M. Wan, and W. G. Halfond, "String analysis for java and android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 661–672, ACM, 2015.

[135] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), p. 468–471, Association for Computing Machinery, 2016.

[136] S. Bhandari, W. B. Jaballah, V. Jain, V. Laxmi, A. Zemmari, M. S. Gaur, M. Mosbah, and M. Conti, "Android inter-app communication threats and detection techniques," *Computers Security*, vol. 70, pp. 392–421, 2017.

[137] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder, "Mr-droid: A scalable and prioritized analysis of inter-app communication risks," in *2017 IEEE Security and Privacy Workshops (SPW)*, pp. 189–198, IEEE, 2017.

[138] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with fuse: Statically detecting android app collusion," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, pp. 1–10, 2014.

[139] K. O. Elish, D. Yao, and B. G. Ryder, "On the need of precise inter-app icc classification for detecting android malware collusions," in *Proceedings of IEEE mobile security technologies (MoST), in conjunction with the IEEE symposium on security and privacy*, 2015.

[140] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *IFIP International Information Security and Privacy Conference*, pp. 513–527, Springer, 2015.

[141] A. Dini Kounoudes, G. M. Kapitsaki, and I. Katakis, "Enhancing user awareness on inferences obtained from fitness trackers data," *User Modeling and User-Adapted Interaction*, pp. 1–48, 2023.

[142] I. Psychoula, L. Chen, and O. Amft, "Privacy risk awareness in wearables and the internet of things," *IEEE Pervasive Computing*, vol. 19, no. 3, pp. 60–66, 2020.

[143] Google, "Google io 2022." December 2022. `https://io.google/2022/`.

[144] G. L. Scoccia, I. Kanj, I. Malavolta, and K. Razavi, "Leave my apps alone! a study on how android developers access installed apps on user's device," in *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pp. 38–49, 2020.

[145] S. Wu and J. Liu, "Overprivileged permission detection for android applications," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2019.

[146] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 217–228, 2012.

[147] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1017–1028, 2013.

[148] B. Michéle and A. Karpow, "Watch and be watched: Compromising all smart tv generations," in *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pp. 351–356, IEEE, 2014.

[149] Y. Bachy, F. Basse, V. Nicomette, E. Alata, M. Kaâniche, J.-C. Courrege, and P. Lukjanenko, "Smart-tv security analysis: practical experiments," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 497–504, IEEE, 2015.

[150] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1332–1349, IEEE, 2020.

[151] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh, and A. Awajan, "Intelligent mobile malware detection using permission requests and api calls," *Future Generation Computer Systems*, vol. 107, pp. 509–521, 2020.

[152] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 252–276, Springer, 2017.

[153] APKMirror, "Apkmirror website." Accessed March 2021. `https://www.apkmirror.com/`.

[154] H. Wang, H. Liu, X. Xiao, G. Meng, and Y. Guo, "Characterizing android app signing issues," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 280–292, IEEE, 2019.

[155] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1039–1055, IEEE, 2020.

[156] S. Sebastian and J. Caballero, "Towards attribution in mobile markets: identifying developer account polymorphism," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 771–785, 2020.

[157] Y. Zhang, S. Ma, T. Chen, J. Li, R. H. Deng, and E. Bertino, "Evilscreen attack: Smart tv hijacking via multi-channel remote control mimicry," *arXiv preprint arXiv:2210.03014*, 2022.

[158] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android custom permissions demystified: From privilege escalation to design shortcomings," *2021 IEEE Symposium on Security and Privacy (SP)*, 2021.

[159] G. S. Tuncay, S. Demetriou, K. Ganju, and C. Gunter, "Resolving the predicament of android custom permissions," in *Network and Distributed System Security Symposium*, Internet Society, 2018.

[160] Google, "Google permissions guide." Accessed March 2021. `https://developer.android.com/guide/topics/permissions/defining`.

[161] P. Grade, "Privacy grade website." Accessed March 2021. `http://privacygrade.org/`.

[162] Vizbee, "Vizbee developers overview." Accessed March 2021. `https://vzb.us/overview/`.

[163] V. Beat, "Android ads triggered up to 21%." Accessed March 2021. https://venturebeat.com/2021/05/19/post-idfa-alliance-finds-ios-14-5-triggered-up-to-21-growth-in-android-ad-spending/.

[164] T. Crunch, "Google removes apps for children." Accessed March 2021. https://techcrunch.com/2020/10/23/google-removes-3-android-apps-for-children-with-20m-downloads-between-them-over-data-collection-violations/.

[165] T. Verge, "Fertility app shared customer data with chinese companies." Accessed March 2021. https://www.theverge.com/2020/8/20/21377591/fertility-app-premom-reportedly-shared-customer-data-with-chinese-companies.

[166] K. von Randow, "Charles proxy homepage." Accessed December 2021. https://www.charlesproxy.com/.

[167] N. Higi, "apk-mitm repository." Accessed December 2021. https://github.com/shroudedcode/apk-mitm.

[168] Google, "Privacy security best practices." Accessed March 2021. https://source.android.com/security/best-practices/privacy.

[169] Google, "Unique identifiers guide." Accessed March 2021. https://developer.android.com/training/articles/user-data-ids.

[170] L. Meftah, R. Rouvoy, and I. Chrisment, "Testing nearby peer-to-peer mobile apps at large," in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 1–11, IEEE, 2019.

[171] TechCrunch, "Disney+ launches its ad-supported tier to compete with netflix." Accessed January 2023. https://techcrunch.com/2022/12/08/disney-launches-its-ad-supported-tier/.

[172] Google, "Android advertising id policy change." Accessed January 2023. https://support.google.com/googleplay/android-developer/answer/6048248.

[173] Google, "Android permissions notes." Accessed March 2021. https://developer.android.com/training/permissions/usage-note.

[174] W. Wang, G. Meng, H. Wang, K. Chen, W. Ge, and X. Li, "A 3 ident: A two-phased approach to identify the leading authors of android apps," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 617–628, IEEE, 2020.

[175] V. Kalgutkar, N. Stakhanova, P. Cook, and A. Matyukhina, "Android authorship attribution through string analysis," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pp. 1–10, 2018.

[176] I. Alam, S. Khusro, and M. Naeem, "A review of smart tv: Past, present, and future," in *2017 International Conference on Open Source Systems & Technologies (ICOSST)*, pp. 35–41, IEEE, 2017.

[177] J. Varmarken, J. Al Aaraj, R. Trimananda, and A. Markopoulou, "Fingerprintv: Fingerprinting smart tv apps," *Proceedings on Privacy Enhancing Technologies*, vol. 3, pp. 606–629, 2022.

[178] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 611–622, 2013.

[179] S. Chaudhary, S. Fischmeister, and L. Tan, "em-spade: a compiler extension for checking rules extracted from processor specifications," *ACM SIGPLAN Notices*, vol. 49, no. 5, pp. 105–114, 2014.

[180] P. Liu, L. Li, Y. Yan, M. Fazzini, and J. Grundy, "Identifying and characterizing silently-evolved methods in the android api. in 2021 ieee/acm 43rd international conference on software engineering: Software engineering in practice (icse-seip 2021)," 2021.

[181] Z. Alyafeai, M. S. AlShaibani, and I. Ahmad, "A survey on transfer learning in natural language processing," *arXiv preprint arXiv:2007.04239*, 2020.

[182] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, (New York, NY, USA), p. 143–153, Association for Computing Machinery, 2019.

[183] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[184] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, pp. 539–546, IEEE, 2005.

[185] W. Yin, J. Hay, and D. Roth, "Benchmarking zero-shot text classification: Datasets, evaluation and entailment approach," *arXiv preprint arXiv:1909.00161*, 2019.

[186] A. Williams, N. Nangia, and S. R. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," *arXiv preprint arXiv:1704.05426*, 2017.

[187] A. Williams, N. Nangia, and S. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language*

Technologies, Volume 1 (Long Papers), pp. 1112–1122, Association for Computational Linguistics, 2018.

[188] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[189] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Cespedes, S. Yuan, C. Tar, *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.

[190] M. E. Peters, S. Ruder, and N. A. Smith, "To tune or not to tune? adapting pretrained representations to diverse tasks," *arXiv preprint arXiv:1903.05987*, 2019.

[191] S. K. Dash, M. Allamanis, and E. T. Barr, "Refinym: Using names to refine types," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, (New York, NY, USA), p. 107–117, Association for Computing Machinery, 2018.

[192] P.-P. Pârtachi, S. K. Dash, M. Allamanis, and E. T. Barr, "Flexeme: Untangling commits using lexical flows," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, (New York, NY, USA), p. 63–74, Association for Computing Machinery, 2020.

[193] J. Kim, J. Jeon, S. Hong, and S. Yoo, "Predictive mutation analysis via the natural language channel in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, jul 2022.

[194] C. Guo, D. Huang, N. Dong, J. Zhang, and J. Xu, "Callback2vec: callback-aware hierarchical embedding for mobile application," *Information Sciences*, vol. 542, pp. 131–155, 2021.

[195] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[196] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, IEEE, 2018.

[197] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *arXiv preprint arXiv:1809.05193*, 2018.

[198] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 631–642, 2016.

[199] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague, "SmartAuth: User-Centered authorization for the internet of things," in *26th USENIX Security Symposium (USENIX Security 17)*, (Vancouver, BC), pp. 361–378, USENIX Association, Aug. 2017.

[200] X. Li, H. Jiang, Y. Kamei, and X. Chen, "Bridging semantic gaps between natural languages and apis with word embedding," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1081–1097, 2018.

[201] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," *Computational Linguistics*, vol. 19, no. 2, pp. 313–330, 1993.

[202] R. Snow, B. O'connor, D. Jurafsky, and A. Y. Ng, "Cheap and fast–but is it good? evaluating non-expert annotations for natural language tasks," in *Proceedings of the 2008 conference on empirical methods in natural language processing*, pp. 254–263, 2008.

[203] Y. Roh, G. Heo, and S. E. Whang, "A survey on data collection for machine learning: A big data - ai integration perspective," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1328–1347, 2021.

[204] J. Zhang, X. Wu, and V. S. Sheng, "Learning from crowdsourced labeled data: a survey," *Artificial Intelligence Review*, vol. 46, pp. 543–576, 2016.

[205] J. C. Chang, S. Amershi, and E. Kamar, "Revolt: Collaborative crowdsourcing for labeling machine learning datasets," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 2334–2346, 2017.