Enhancing Users' Privacy: Static Resolution of the Dynamic Properties of Android

by

Abhishek Tiwari

Doctoral thesis

submitted in fulfillment of the requirements for the degree of Doctor of Engineering (Dr.-Ing.) in Software Engineering

Faculty of Science

Institute of Computer Science

Software Engineering Group

University of Potsdam

Supervisor: Prof. Dr.-Ing., Christian Hammer

Mentor: Prof. Dr., Christoph Kreitz

Reviewers: (1) Prof. Dr.-Ing., Christian Hammer,

(2) Asst. Prof. Dr., Alessandra Gorla,

(3) Dr.-Ing., Sven Buigel

Place and Date of the Disputation: Potsdam, 27.06.2019

Abstract

The usage of mobile devices is rapidly growing with Android being the most prevalent mobile operating system. Thanks to the vast variety of mobile applications, users are preferring smartphones over desktops for day to day tasks like Internet surfing. Consequently, smartphones store a plenitude of sensitive data. This data together with the high values of smartphones make them an attractive target for device/data theft (thieves/malicious applications).

Unfortunately, state-of-the-art anti-theft solutions do not work if they do not have an active network connection, e.g., if the SIM card was removed from the device. In the majority of these cases, device owners permanently lose their smartphone together with their personal data, which is even worse.

Apart from that malevolent applications perform malicious activities to steal sensitive information from smartphones. Recent research considered static program analysis to detect dangerous data leaks. These analyses work well for data leaks due to inter-component communication, but suffer from shortcomings for inter-app communication with respect to precision, soundness, and scalability.

This thesis focuses on enhancing users' privacy on Android against physical device loss/theft and (un)intentional data leaks. It presents three novel frameworks: (1) ThiefTrap, an anti-theft framework for Android, (2) IIFA, a modular inter-app intent information flow analysis of Android applications, and (3) PIAnalyzer, a precise approach for PendingIntent vulnerability analysis.

ThiefTrap is based on a novel concept of an anti-theft honeypot account that protects the owner's data while preventing a thief from resetting the device. We implemented the proposed scheme and evaluated it through an empirical user study with 35 participants. In this study, the owner's data could be protected, recovered, and anti-theft functionality could be performed unnoticed from the thief in all cases.

IIFA proposes a novel approach for Android's inter-component/inter-app communication (ICC/IAC) analysis. Our main contribution is the first fully automatic, sound, and precise ICC/IAC information flow analysis that is scalable for realistic apps due to modularity, avoiding combinatorial explosion: Our approach determines communicating apps using short summaries rather than inlining intent calls between components and apps, which requires simultaneously analyzing all apps installed on a device. We evaluate IIFA in terms of precision, recall, and demonstrate its scalability to a large corpus of real-world apps. IIFA reports 62 problematic ICC-/IAC-related information flows via two or more apps/components.

PIAnalyzer proposes a novel approach to analyze PendingIntent related vulnerabilities. PendingIntents are a powerful and universal feature of Android for inter-component communication. We empirically evaluate PIAnalyzer on a set of 1000 randomly selected applications and find 1358 insecure usages of PendingIntents, including 70 severe vulnerabilities.

Zusammenfassung

Die Nutzung von mobilen Geräten nimmt rasant zu, wobei Android das häufigste mobile Betriebssystem ist. Dank der Vielzahl an mobilen Anwendungen bevorzugen Benutzer Smartphones gegenüber Desktops für alltägliche Aufgaben wie das Surfen im Internet. Folglich speichern Smartphones eine Vielzahl sensibler Daten. Diese Daten zusammen mit den hohen Werten von Smartphones machen sie zu einem attraktiven Ziel für Geräte/Datendiebstahl (Diebe/bösartige Anwendungen).

Leider funktionieren moderne Diebstahlsicherungslösungen nicht, wenn sie keine aktive Netzwerkverbindung haben, z. B. wenn die SIM-Karte aus dem Gerät entnommen wurde. In den meisten Fällen verlieren Gerätebesitzer ihr Smartphone dauerhaft zusammen mit ihren persönlichen Daten, was noch schlimmer ist.

Abgesehen davon gibt es bösartige Anwendungen, die schädliche Aktivitäten ausführen, um vertrauliche Informationen von Smartphones zu stehlen. Kürzlich durchgeführte Untersuchungen berücksichtigten die statische Programmanalyse zur Erkennung gefährlicher Datenlecks. Diese Analysen eignen sich gut für Datenlecks aufgrund der Kommunikation zwischen Komponenten, weisen jedoch hinsichtlich der Präzision, Zuverlässigkeit und Skalierbarkeit Nachteile für die Kommunikation zwischen Apps auf. Diese Dissertation konzentriert sich auf die Verbesserung der Privatsphäre der Benutzer auf Android gegen Verlust/Diebstahl von physischen Geräten und (un)vorsätzlichen Datenverlust. Es werden drei neuartige Frameworks vorgestellt: (1) ThiefTrap, ein Anti-Diebstahl-Framework für Android, (2) IIFA, eine modulare Inter-App Analyse des Informationsflusses von Android Anwendungen, und (3) PIAnalyzer, ein präziser Ansatz für PendingIntent Schwachstellenanalyse.

ThiefTrap basiert auf einem neuartigen Konzept eines Diebstahlschutzkontos, das die Daten des Besitzers schützt und verhindert, dass ein Dieb das Gerät zurücksetzt. Wir haben das vorgeschlagene Schema implementiert und durch eine empirische Anwenderstudie mit 35 Teilnehmern ausgewertet. In dieser Studie könnten die Daten des Besitzers geschützt und wiederhergestellt werden, und die Diebstahlsicherungsfunktion konnte in jedem Fall unbemerkt vom Dieb ausgeführt werden.

IIFA schlägt einen neuen Ansatz für die Analyse von Komponenten zwischen Komponenten/ Inter-App Kommunikation (ICC/IAC) von Android vor. Unser Hauptbeitrag ist die erste vollautomatische, solide und präzise ICC/IAC Informationsflussanalyse, die aufgrund ihrer Modularität für realistische Apps skalierbar ist und eine kombinatorische Explosion vermeidet: Unser Ansatz bestimmt, dass Apps über kurze Zusammenfassungen kommuniziert werden, anstatt Absichtsaufrufe zwischen Komponenten zu verwenden und Apps, bei denen gleichzeitig alle auf einem Gerät installierten Apps analysiert werden müssen. Wir bewerten IIFA in Bezug auf Präzision, Rückruf und demonstrieren seine Skalierbarkeit für einen großen Korpus realer Apps. IIFA meldet 62 problematische ICC- / IAC-bezogene Informationsflüsse über zwei oder mehr Apps / Komponenten.

PIAnalyzer schlägt einen neuen Ansatz vor, um Schwachstellen im Zusammenhang mit PendingIntent zu analysieren. PendingIntents nutzen eine leistungsstarke und universelle Funktion von Android für die Kommunikation zwischen Komponenten. Wir evaluieren PIAnalyzer empirisch an einem Satz von 1000 zufällig ausgewählten Anwendungen und finden 1358 unsichere Verwendungen von PendingIntents, einschließlich 70 schwerwiegender Schwachstellen.

Contributions of this Dissertation

This dissertation focuses on improving users' privacy in Android operating system. This dissertation would not have been possible without the guidance and valuable feedback of my supervisor (Prof.Dr.-ing. Christian Hammer). The **major contributions** are as follows:

The author and Sascha Groß are the two main co-authors of ThiefTrap [39]. The author and Sascha Groß had the initial idea and motivation for a deceiving GUI as an anti-theft measure. The author and Sascha Groß transformed the initial idea to a working concept of ThiefTrap on Android operating system. The author was responsible for configuring the Android open source project, implementing fake factory reset scenarios on both boot loader and Settings menu, and the implementation of some fake settings required after the fake factory reset. Additionally, the author along with Sascha Groß came up with the suitable evaluation criteria required for empirically evaluation of ThiefTrap. All authors performed reviews of the paper.

The IIFA [82] concept was developed in a joint effort between the author and Sascha Groß. The author was further responsible for the implementation and evaluation of IIFA. The author took great effort in implementing all aspects of the problem (Wrote approximately 8k lines of code). The author empirically evaluated IIFA on three state-of-the-art benchmarks, compared it with six state-of-the-art related tools in terms of precision and recall. Additionally, the author evaluated IIFA's output with state-of-the-art related tools and demonstrated IIFA's effectiveness on real world applications. All authors performed reviews of the paper.

The author and Sascha Groß are the two main co-authors of PIAnalyzer [38]. The author and Sascha Groß created the attack scenario of PendingIntent. The author was further responsible for evaluating PIAnalyzer. All authors performed reviews of the paper.

Minor Contributions

The IFC for Hybrid Android Applications [41] was developed in a joint effort between the author and Enrico Höschler. The author came up with the initial idea and motivation of the work. The author further designed the methodology and the evaluation criteria. Enrico Höschler was involved in major parts of the implementation and evaluation.

The author upgraded [81] the DexLib (from version 1 to 2) in the widely used framework WALA [20], to facilitate the analysis of newer Android applications.

Acknowledgment

At this point, I would like to thank God and all those who have helped me in the completion of this thesis. This work would not have been possible without their help and support.

First, I would like to thank my primary advisor, Christian Hammer, for his endless support and guidance. The quality of feedback and ideas that he provides have reshaped my thinking process towards solving a problem and made me a better researcher. Apart from the academic support, he also helped me in several bureaucratic procedures. Additionally, I admire him for providing such an energetic and stress-free research environment.

I would like to thank my closest co-author Sascha Groß, without whom this thesis would not have been possible. We had a lot of fruitful discussions regarding research and on general topics. He has always helped me in handling all the German documents and translations. Additionally, I would like to thank all of my colleagues for providing a positive working environment.

I will like to thank all of my friends, especially Sumit Shekhar, Manish Mishra and Jyoti Prakash, for their continuous support. They have always helped me to overcome my descents and provided me their valuable advice for important decisions.

Last but not the least, I am extremely grateful to my family especially my father Radha Mohan Tiwari and my mother Sheela Tiwari, for their unconditional love and support. Without them, I would certainly not stand where I stand today.

Contents

Part I

Introduction and Background

Chapter 1

Introduction

The usage of mobile devices is rapidly growing with Android being the most prevalent mobile operating system (with a global market share of 72.23% as of November 2018 [40]). Various reports [46, 17] reveal that the mobile application (app) usage is growing by 6% year-over-year and users are preferring mobile apps over desktop apps. As is, mobile devices store a plenitude of sensitive data, including sensitive personal or financial information as well as session tokens of online services used in installed applications. This data together with the high values of smartphones make them an attractive target for physical theft. Clearly, the device owner would like to regain the device in such a case. Additionally, if this data is leaked to a malevolent entity (a malicious application) these tokens can be leveraged to perform unauthorized actions on the victim's behalf, potentially leading to severe harm for the device owner. Therefore, protecting the information stored on smartphones from unauthorized access has become imperative.

In this thesis, we propose novel frameworks to protect the user's privacy/data on Android operating system. In general we answer to the following research questions: **RQ1**. How to protect user's privacy/data against physical device loss/theft? **RQ2.** How to protect user's privacy/data against (un)intentional application leaks?

To protect against physical device loss/theft, there were two possible mechanisms: *1. Anti-theft applications:* Most of the anti-theft applications provide the functionality to lock the phone, erase it or triggering an alarm from remote. Unfor-

tunately, anti-theft applications do not work if they do not have an active network connection e.g., if the SIM card was removed from the device. *2. Google Device Protection:* Starting from Android version 5.1, Google released a new feature called "Device Protection" [68]. This anti-theft feature makes it impossible for a thief to use a stolen phone after it was factory reset. However, this feature is only available for a small number of devices, which are capable of running Android version 5.1 or greater. More than 45% of the Android phones use a lower Android version [32]. Moreover, this mechanism gives no profit to the thief, but the device owner still losses the device and/or his personal data.

To protect against (un)intentional application leaks on Android, various information flow control (IFC) analyses [5, 43, 50, 84, 53, 7] have been developed. These approaches analyze the (potential) flow of information in apps and report a warning if a flow from a sensitive data source to an untrusted/public data sink (like sending sensitive information to the internet) is determined to be possible at runtime. Information flow is not restricted to a single component, but occurs frequently between components of the same [53, 36] and even different apps [84]. Our study using the top 90 apps from the Google play store revealed more than 10,000 inter-component calls. Scrutinizing the flows between components therefore becomes imperative. While state-of-the-art information flow analyses handle the information flow well inside one app, they suffer from several shortcomings with respect to precision, soundness, and scalability in case of inter-app communication.

Further, in our study we discover that the Android permission system can be circumvented in many cases in the form of denial-of-service, identity theft, and privilege escalation attacks. By exploiting vulnerable but benign applications that are insecurely using PendingIntents, a malicious application *without any permissions* can perform many critical operations, such as sending text messages (SMS) to a premium number. PendingIntents are a widespread Android callback mechanism and reference token. While the concept of PendingIntents is flexible and powerful, insecure usage can lead to severe vulnerabilities. Yu et al. [83] report a PendingIntent vulnerability in Android's official Settings app, which made a privilege escalation attack up to SYSTEM privileges possible for every installed application. Thus, given the severe security implications, the official Android documentation on PendingIntents [27] now warns against insecure usage. However, to the best of our knowledge, to-date no analysis tool detects the described PendingIntent vulnerabilities. Thus,

an automated analysis tool is envisioned that scales to a large number of applications.

Our Contributions:

Our work on enhancing users' privacy on Android comprises of the following publications [39, 82, 38], each of which contribute to the development of a secure Android operating system as presented in this thesis:

ThiefTrap. [39] To answer the RQ1, we propose ThiefTrap, an anti-theft framework that uses the Android account feature as a security measure to protect against device thieves. We are the first to use this feature in that we set up a honeypot account simulating the device owner's account. In this mechanism the device owner has the option to configure an anti-theft honeypot account, which resembles the owner's regular account except for some modifications. A person that is not the device owner can never distinguish interacting with the anti-theft account or the real account. The anti-theft account hides the personal data of the device owner and performs hidden anti-theft functionality while the thief uses the device. One important feature of this framework is that when the thief performs a factory reset, our approach only gives the illusion that a factory reset is being done, while in reality all of the owner's data is preserved. After a fake factory reset the device hides the owner's account completely but still executes the anti-theft functionality. The thief will then start using the smartphone as a new device, unaware of anti-theft functionality executing in the background. It is likely that after the fake factory reset a thief will establish an internet connection by inserting a SIM card or establishing a WIFI connection. At this point of time the hidden anti-theft functionality can for example send identifying information of the thief to the device owner or start listening for remote commands from the device owner. So the device owner likely will be able to recover the device and the personal data. The key benefit of this approach is that it improves chances of identifying the thief and regaining the stolen phone as well as the personal data.

We evaluated our approach in the form of an empirical user study. Our study with 35 participants, showed that in all cases our approach prevented loss of ***owner's personal data*** and performed the ***required anti-theft functionality***. In the very vast majority of cases the potential thief was completely oblivious to our approach.

To answer the RQ2, we provide the following frameworks:

IIFA. [82] IIFA is a novel information-flow analysis for IAC (and ICC) based on an intent-flow pre-analysis that evades combinatorial explosion of analyzing all potential communication partners, while precisely matching type and key information of intent data. Our approach can predict which combinations of apps communicate by separating information flow analysis within app components and matching of communication partners. In a first step we create a database of summary information about senders of intents, their characteristics including types and keys, and outbound intent data, as well as apps registered to receive certain implicit intents. This information can then be matched in a subsequent step to identify potential communication partners. Further we propose a novel matching algorithm, based on a baseline IFC analysis providing potential intra-component flows (including a program slices of the receiver's key value) for all potential intent receivers. We leverage senders' outbound intent data as input to the information flows identified in respective receivers, which eliminates the need for inlining or merging apps and thus combinatorial explosion, as only summaries of actual communication partners are subsumed. In case multiple apps are involved in intent communication our approach performs a light-weight fixed point iteration through the DB information. Note that our tool is not a stand-alone IFC analysis tool. Rather, IIFA leverages flows and slices generated by other IFC analyzers. As these tools are already heavily engineered for the intra-app case, we concentrated on the peculiarities of intent communication and evasion of inlining and combinatorial explosion.

As a noteworthy novelty, our approach is modular and thus compositional with respect to app installation. Whenever a new (version of an) app is available for analysis, the database is updated (in case of new version) or extended (new app) to include the intents broadcast or received by this app. Only the new app has to be (re-)analyzed, as well as combinations with flows identified in potential receivers. We compute precise communication paths between components, handling complex control flows such as in callback methods (see section 4.4.2). As intent targets are specified via a string parameter, our approach can resolve common string manipulations, which improves our precision significantly for regular (non-obfuscated) apps. As a minor contribution we took great effort to handle the full spectrum of intent communication, supporting explicit and implicit intents as well as dynamically created intent receivers. All previous approaches miss at least one of these

features, leading to unsoundness and imprecision. Our analysis is fully automated and does not require the source code of the app under analysis. We aim to answer the following research questions:

- *Does our pre-analysis approach negatively impact the precision or soundness of the results with respect to state-of-the-art analyses?*
- *Does our approach scale to a realistic corpus of real-world apps?*
- *How do common real-world apps communicate through IAC?*

We evaluated IIFA on DroidBench [18], the IccTA extension on DroidBench, ICC Bench, our own benchmarks evaluating key and type matching of intent extra data, and a large set of apps from the Google Playstore. We compared our results with multiple related analysis tools. Our tool (combined with an external baseline intra-component IFC analysis) achieves perfect precision and soundness on all benchmark sets, being more than on par with related IFC tools that consider intent communication. Additionally, we demonstrate that IIFA can improve the IAC precision of other base IFC analyses with experiments, and assess the scalability of IIFA, applying it to the 90 most downloaded Playstore apps. Our experiments demonstrate that due to its compositionality IIFA's execution time scales well even to a large corpus of real-world apps.

PIAnalyzer. [38] PIAnalyzer is a novel approach and a tool to detect PendingIntent related vulnerabilities in Android applications. In multiple analysis steps, PIAnalyzer computes the relevant information of the potentially vulnerable code based a program slicing [85]. PIAnalyzer is fully automated and does not require the source code of the application under inspection. PIAnalyzer assists human analysts by computing and presenting vulnerability details in easily understandable log files. We evaluated PIAnalyzer on 1000 randomly selected applications from the Google Play Store. We discover 435 applications that wrap at least one implicit base intent with a PendingIntent object, out of which 1358 insecure usages of PendingIntents arise. These include 70 PendingIntent vulnerabilities leading up to the execution of critical operations from unprivileged applications. We manually investigate multiple findings and inspect reports on examples known to be vulnerable. Our investigation show that PIAnalyzer is highly precise and sound.

Chapter 2

Background

2.1 Android Framework

Android is the world's most popular mobile operating system. Figure 2.1 depicts the internal structure of Android OS.

The Android framework can be best described in the form of different layers. The lowest layer, a customized *Linux Kernel*, is used for drivers and hardware support. The subsequent hardware abstraction layer (HAL) provides a standard interface for exposing the hardware capabilities to the higher-level Android frameworks. HAL implementations are built into shared library modules (.so files).

Android applications are compiled to a specific bytecode format (DEX) designed specially for Android. The Android runtime (ART) provides a Dalvik virtual machine, which is similar to the standard Java virtual machine, but designed and optimized for Android.

The Android framework layer provides many higher-level services in the form of an API to the Application layer. These APIs act as the building blocks to create Android applications. Application developers utilize these APIs in their applications. Most of our changes are implemented in this layer.

The topmost layer, the application layer, provides different applications to be used by end users, such as alarms, browser, calculator etc. Google provides a cen-

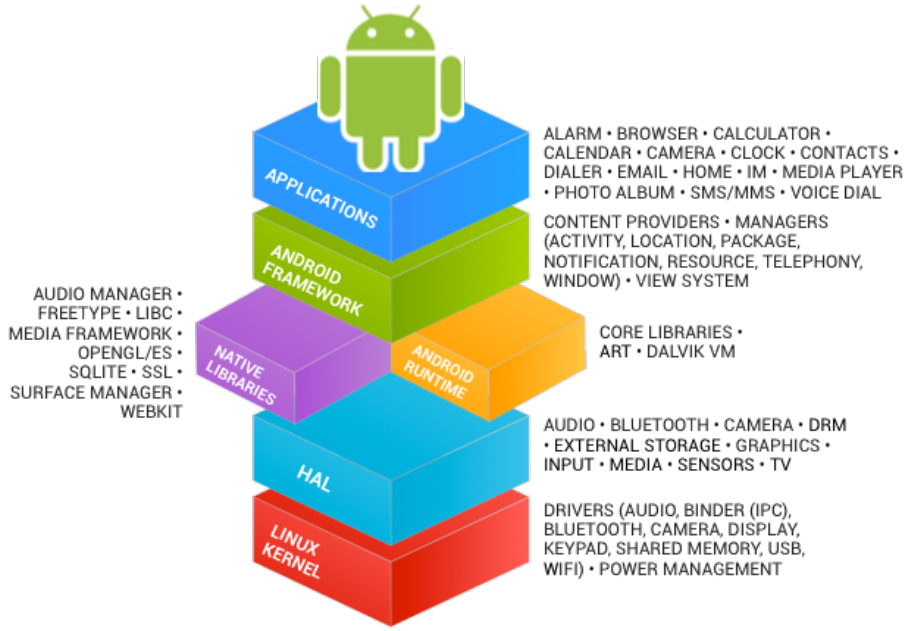


Figure 2.1: Android Architecture [30]

tral store, for developers to publish their applications, called the Google play store. As of December 2016, the Google play store included over 2.5 million apps.

2.2 Anti-theft Mechanisms

Anti-theft mechanisms are supposed to prevent device theft or mitigate the damage in case a device gets stolen or is lost. One kind of anti-theft solutions tracks information of the device after it was stolen and provides the information to the device owner. Several anti-theft solutions rely on providing location information and remote administration functionality to the device owner. Some of them also provide the possibility to recover personal data or remotely wipe the device. At present, there are two options for Android owners to protect their device against theft. The first option is to use the Android device theft protection feature (available for devices capable of supporting Android version greater than 5.0). The second option is to use an third party anti-theft application.

- *Android built-in anti-theft mechanisms:* Starting from Android version 5.1, Google released a new feature called "Android Device Protection" [86, 68]. This anti-theft feature prevents a thief from using a stolen phone that has been wiped. However, more than 45% of the Android phones use a lower Android version

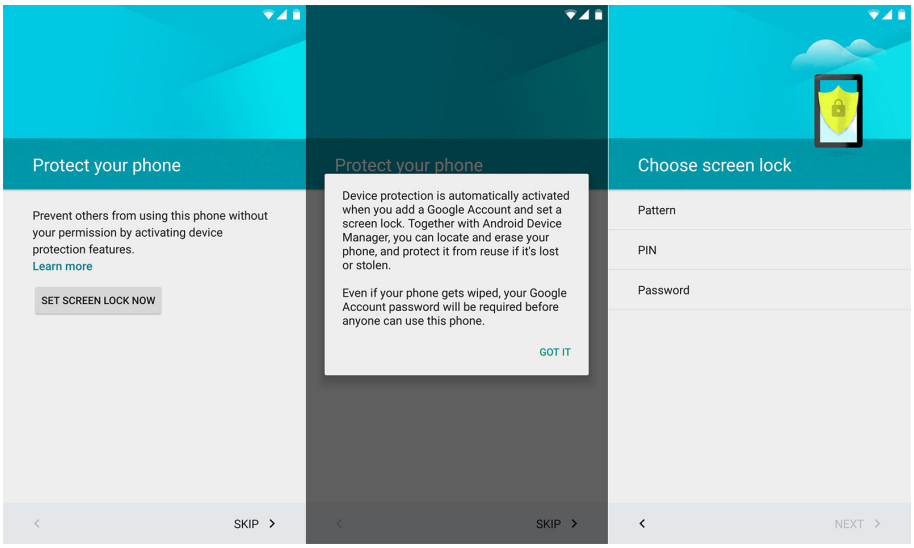


Figure 2.2: The Android Device Protection feature [86]

[32]. In addition, this feature will not work without a proper setup. For example, the user needs to log into a Google account on the device. Then, if a device supports this feature, Android Device Protection is enabled as soon as the user enables a locking mechanism. Figure 2.2 explains the activation of this feature while enabling a device locking mechanism.

After a factory reset, Android Device Protection requires the user to enter the Google account credentials on which the device was previously configured. This renders the device unusable to the thief even after a factory reset was performed. However, an unlocked bootloader still allows to flash a binary on the device, thus this feature is not available for devices with an unlocked bootloader.

- *Anti-theft Applications:* There is a multitude of third-party anti-theft applications available in the app stores. These applications provide features like locating the device, remotely administrating the device etc.

  The anti-theft features of these applications heavily rely on a network connection. These applications are mostly not functional if the SIM card was removed from the device. A thief is likely to remove the SIM card from a stolen device and to turn it off, such that the device loses its network connections. Later, in order to reconfigure the device as new, a thief is likely to perform a factory reset, so the anti-theft application is removed from the device, leaving it unprotected. Additionally, the personal data of the device owner is lost

unrecoverable.

2.3 Android Applications

Android applications are written Kotlin [10], Java and C++. Instead of defining a *main* method they may consist of four component types: Activities, Services, Broadcast Receivers and Content Providers. Activities are user interfaces a user can interact with. Services run in the background and are intended for computationally expensive operations. Broadcast receivers are components that register to receive system or application events. Content providers provide data for an application via storage mechanisms. The message passing mechanism on Android is inter-component communication via so-called intents. Each application in Android needs to define a manifest file (*AndroidManifest.xml*). The manifest file provides essential information about the application, e.g., which resources an application may access. Components and their capabilities are defined in the manifest file.

Android applications are compiled from Java source code to Dalvik bytecode [26] which is specialized for execution on Android. Finally, the compiled classes are compressed into an Android Package (APK) together with additional metadata and resources. APKs are usually published in market places, such as the Google Playstore. Oberheide and Miller [60] showed that Google's security analysis for Android applications can easily be circumvented. Even though Google's security mechanisms are constantly improving, there is still a big potential for malicious and vulnerable software in the Google Playstore.

As mentioned, in the first compilation step Android applications are compiled from Java source code to Dalvik bytecode. This bytecode is hard to read and analyze for humans. We use *Apktool* [1] to compile the bytecode into the Smali [23] format, an intermediate representation of Dalvik bytecode. Smali improves code readability and eases analysis and modification.

2.3.1 Android Intents

Android provides a dedicated mechanism to communicate between different components, called intent. An intent is a message that is sent from one component to another, for example, to notify another component of an event, trigger an action

of another component, or transmit information from one component to another. One should hereby notice the universal usage of intents in Android: Intents can be sent from the system to applications (and vice versa), from one application to another (inter-app communication), or even from one component to another within the same app (intra-app-communication) [25].

Additional information can be associated with an intent, namely an intent action, a target component, and intent extra data. The intent action specifies the action that is supposed to be performed by the receiving component. To receive a specific intent action, the receiving component needs to declare it in the manifest file via an intent filter (see discussion of Listing 4.2 below). On the receiver side, the sender of an intent is unknown. The target component specifies a particular receiver for the intent. Setting intent extra data adds additional information to be used by the receiving component.

It is important to remark that none of these pieces of information is mandatory. When the target component is set one calls an intent *explicit*, otherwise *implicit*. The important difference is that explicit intents are only delivered to the defined target component, while implicit intents can be delivered to every component with a matching intent filter . If multiple components can receive the intent, the user is asked to resolve the intent manually, generally displaying a list of potential receiver apps. Li et al. [53] found that at runtime 40.1% of the intents in Google Playstore applications are explicit intents. A third category of intents is broadcast intents, which are broadcast to every registered component for an intent action instead of only one. Finally, the last category of intents are pending intents. Pending intents are intended for giving another application the possibility to perform a certain action in the context of the sending application. The usage and security implications of pending intents will be discussed below.

2.3.2 PendingIntent

A PendingIntent is a special kind of intent which stores a base intent that is to be executed at some later point in time by another component/application, but with the original app's identity and permissions. The point is that the original app is not required to be in memory or active at that point of time, as the receiver will execute it as if executed by the original application. Thus PendingIntent is applicable

```
1 //Component A: Create the base Intent with a target component
2   Intent baseIntent = new Intent("TARGET_COMPONENT");
3 //Create a PendingIntent object wrapping the base Intent
4   PendingIntent pendingIntent = PendingIntent.getActivity(this, 1,
      baseIntent, PendingIntent.FLAG_UPDATE_CURRENT);
5 //Component B (may be in another application or within a system
      manager):
6 //Execute the PendingIntent (internally launches the base intent)
7  try {
8     pendingIntent.send();
9 } catch (PendingIntent.CanceledException e) {}
```

Listing 2.1: A simple PendingIntent Usage

in cases where normal intents are not. "A PendingIntent itself is simply a reference to a token maintained by the system describing the original data used to retrieve it. This means that even if its owning application's process is killed, the PendingIntent itself will remain usable from other processes that have been given it" [27]. A possible usage scenario for PendingIntent is a notification. If an application wishes to get notified by the system at a later point of time, it can create a PendingIntent and pass this PendingIntent to the Notification Manager. The Notification Manager will trigger this PendingIntent in the future, and so a predefined component of the application will be notified and gets executed.

Programmatically, the usage of a PendingIntent is a three step process (Listing 2.1). First, the so called *base intent* is created. The base intent is an ordinary intent which defines the action to be performed on the execution of the PendingIntent. The PendingIntent object wraps the base intent using the factory methods *getActivity()*, *getActivities()*, *getBroadcast()* or *getService()*. These factory methods define the nature of the base intent, e.g., *PendingIntent.getBroadcast()* will launch the base intent as a broadcast intent. The PendingIntent object returned by these methods can be passed to another application or system component, e.g., it can be embedded in another intent object (the wrapping intent) as extra data to make it available to other applications. It is also common to pass a PendingIntent object to a system component, e.g., the AlarmManager, for callback purposes.

Security Implications: Whenever a PendingIntent is triggered, the associated base intent is executed in the context (with the same privileges and name) of the application that created it. However, the three main pieces of data of the base intent may be changed even after the PendingIntent has been handed to another compo-

nent, which may alter the semantics of the base intent that is to be executed with the original app's identity and permissions. While the Target Component or Action of the base Intent cannot be overridden by an attacker if already defined by the sender, an undefined Action or Target Component may be defined after handing it to the receiver. Finally, extra data, which is effectively a key-value store, can always be added after the fact. The implications include that an implicit intent (with no target component defined) can be altered by the receiving app to target any component it desires (and which the original app's permission support), including system features like wiping the phone.

As a consequence, the Android documentation of PendingIntent [27] explicitly warns about potential vulnerabilities caused by misusage: "By giving a PendingIntent to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity) (but just for a predefined piece of code). As such, you should be careful about how you build the PendingIntent: almost always, for example, the base Intent you supply should have the component name explicitly set to one of your own components, to ensure it is ultimately sent there and nowhere else." In fact, if a malicious application can retrieve a PendingIntent from another application with an implicit base intent, it may perform a restricted form of *arbitrary code execution* in the context of the application that created the PendingIntent object: As many (but not all) permission-clad functionalities are accessible via intents, the attacker can reroute the base intent to such functions.

Part II

Privacy/Data Protection Against Physical Device loss/theft

Chapter 3

ThiefTrap: An Anti-Theft Framework for Android

3.1 Overview

Smartphones play a vital role in everyone's life. Their contribution is significant in every day to day activity. Nowadays, smartphones are used for various activities such as capturing pictures, browsing the internet, and using online banking. However, these great advantages come at a price. If the device gets in the wrong hands, the device owner does not only lose the device but also a great amount of personal data. A thief getting hold of personal data and trying to exploit it may result in fraud or blackmailing. It is of utmost importance to provide some mechanism to protect the theft of smartphone devices and the personal data on them. These device protection mechanisms are called *anti-theft mechanisms*.

At present, the smartphone market ranges from around 50 USD to 1000 USD. Loss or theft of a phone does not only result in financial deprivation but also of the personal data which is stored on the device. According to a study [67], the number of stolen smartphones rose to 3.1 million in 2013. Another study [45] reveals that victims are willing to pay 500 to 1000 USD to regain their personal data including photos and videos.

The Android market share is continuously increasing [44] and it dominates the

smartphone market with a share of 86.8%, by the end of Q3 2016. Taking this into the consideration, we targeted the Android platform for the implementation of our approach. At present, there are two possible anti-theft mechanisms:

1. *Anti-theft applications:* Most of the anti-theft applications provide the functionality to lock the phone, erase it or triggering an alarm from remote. Unfortunately, anti-theft applications do not work if they do not have an active network connection e.g., if the SIM card was removed from the device.

2. *Google Device Protection:* Starting from Android version 5.1, Google released a new feature called "Device Protection" [68]. This anti-theft feature makes it impossible for a thief to use a stolen phone after it was factory reset. However, this feature is only available for a small number of devices, which are capable of running Android version 5.1 or greater. More than 45% of the Android phones use a lower Android version [32].

In the majority of these cases, device owners permanently lose their smartphone together with their personal data, which is even worse. As a remedy, we propose a mechanism where a device owner has the option to configure an anti-theft honeypot account, which resembles the owner's regular account except for some modifications. A person that is not the device owner can never distinguish interacting with the anti-theft account or the real account. The anti-theft account hides the personal data of the device owner and performs hidden anti-theft functionality while the thief uses the device. One important feature of this framework is that when the thief performs a factory reset, our approach only gives the illusion that a factory reset is being done, while in reality all of the owner's data is preserved. After a fake factory reset the device hides the owner's account completely but still executes the anti-theft functionality. The thief will then start using the smartphone as a new device, unaware of anti-theft functionality executing in the background. It is likely that after the fake factory reset a thief will establish an internet connection by inserting a SIM card or establishing a WIFI connection. At this point of time the hidden anti-theft functionality can for example send identifying information of the thief to the device owner or start listening for remote commands from the device owner. So the device owner likely will be able to recover the device and the personal data. The key benefit of this approach is that it improves chances of identifying the thief and regaining the stolen phone as well as the personal data.

3.2 Our Contributions

We propose ThiefTrap, an anti-theft framework that uses the Android account feature as a security measure to protect against device thieves. We are the first to use this feature in that we set up a honeypot account simulating the device owner's account. Technically, we provide the following contributions:

1. *ThiefTrap.* We propose ThiefTrap, a novel concept, using a honeypot account for the purpose of theft protection. This concept is the first anti-theft solution that at the same time protects the confidentiality of the owner's user data, prevents loss of this data and provides the full functionality of every anti-theft solution. An important benefit of the proposed approach compared to existing anti-theft solutions is that a device instrumented with our approach is *indistinguishable* from an ordinary device. Our approach ensures that the device and the personal data on it can be regained with high probability.

2. *Implementation.* We implemented our concept in the latest version of Android (7.1_r1 Nougat) of the Android Open Source Project.

3. *Evaluation.* We evaluated our approach in the form of an empirical user study. Our study with 35 participants, showed that in all cases our approach prevented loss of ***owner's personal data*** and performed the ***required anti-theft functionality***. In the very vast majority of cases the potential thief was completely oblivious to our approach.

3.3 Methodology

When a device is stolen there are two possibilities depending on whether the device is protected by a locking mechanism or not. In case the device is not protected by a locking mechanism, a thief immediately has unlimited access to the device owner's data and may result in abuse of the user data on the device (e.g. credentials) to inflict further harm to the device owner. If the device is locked, it is of no use to the thief as long as the locking mechanism is in place. For this reason it is likely that the thief will factory reset the device, in which case all user data on the device is ultimately lost. Modern smartphones store a lot of valuable private data. Additionally, installed anti-theft applications will be removed from the device so chances are

minimal that the device can be retrieved by the device owner. Both of these scenarios are unsatisfying. Therefore, there exists a need for a solution that protects the confidentiality of the user data, while it prevents the device from being factory reset illegitimately. In this work we propose the first approach that can protect the confidentiality of the device owner's user data, while preventing a thief from factory resetting the device and thus removing installed anti-theft applications.

We propose the concept of a honeypot account for theft protection. We leverage the Android guest account feature to implement the honeypot account. Android's multi-account feature was introduced in Android version 4.2. According to statistics provided by Google [33] , this feature is supported by 95.8% of devices. The idea of this concept is that it pretends to be the account of the device owner, while it actually is an isolated honeypot account and prevents any access to the user data of the device owner. A thief logging into the device using the home button or power on button, is logged into the honeypot account, which pretends to be the device owner's account. Therefore the honeypot account tricks a thief into believing that he/she is interacting with an unprotected device in an ordinary way, while actually interacting with the honeypot account. The device owner can log into the real account using a hidden mechanism e.g., using fingerprint lock. This mechanism can be configured by the device owner.

Using the proposed concept of a honeypot account, the privacy of the user data is protected. At the same time, in our approach a factory reset initiated from the honeypot account is simulated s.t. the thief believes that the factory reset is being performed, while in reality the owner's data is preserved. After this simulated factory reset, the thief is presented a new account as expected. However, this new account is a customized honeypot account, which runs an anti-theft mechanism in the background hidden from the thief. The great strength of this concept becomes notable when it is combined with existing anti-theft solutions. As the device owner knows that a honeypot account is installed on the phone, he/she will not log into the honeypot account of that device. For this reason it is likely that a user interacting with the honeypot account for some time is not authorised to do so. Therefore, in our approach an anti-theft solution is installed and will be triggered whenever an user interacts with the honeypot account for some time. This anti-theft solution can then for example collect data of the thief and send them to the device owner, who can use them for regaining the device. Figure 5.1 shows the described workflow.

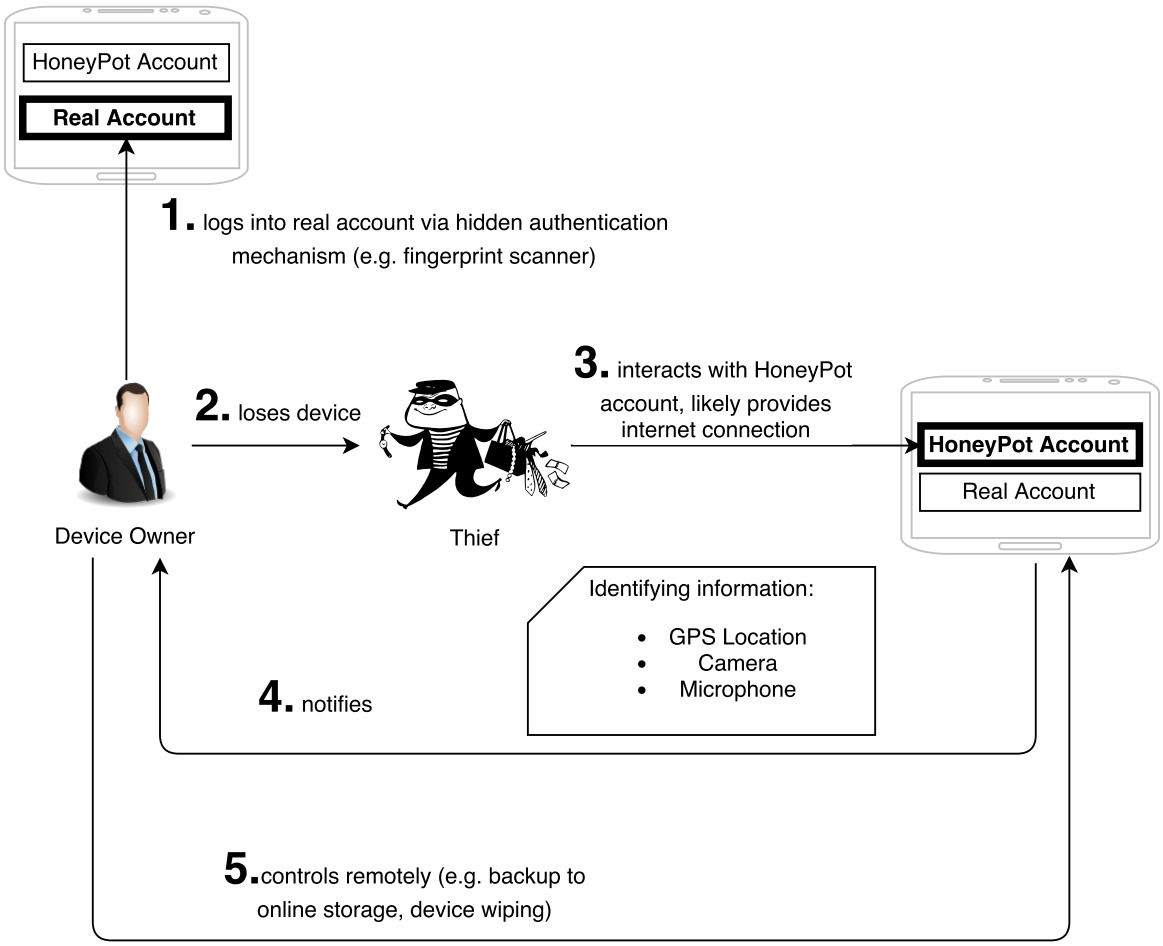
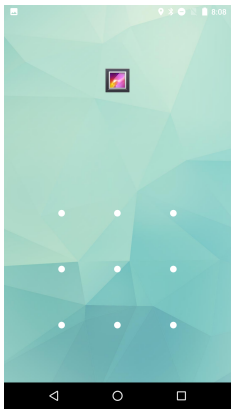


Figure 3.1: Workflow of a device instrumented by ThiefTrap

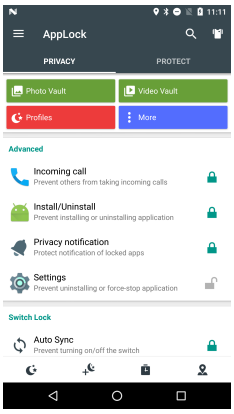
We would like to stress that there exist numerous ways to implement the concept of an anti-theft honeypot-account on various platforms. We choose to implement our approach on the Android operating system. We use the uncustomized version 7.1_r1 from the Android Open Source Project [31]. At the time of this writing, Android is the most used mobile operating system with over 1.4 billion devices in usage and version 7.1_r1 is the latest version of Android.

3.3.1 HoneyPot Account, A Simulation of the Owner's Account

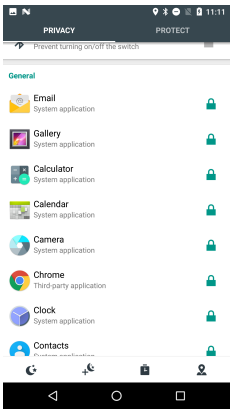
The honeypot account is an Android's empty guest account with some modifications. The idea of the honeypot account is to deceive the thief into the belief that he/she is interacting on the real account. Therefore, it is important to provide the illusion of user data in the honeypot account. In principle any concept for simulating user data can be used. In our approach, the applications in the honeypot account



(a) AppLock protecting the Gallery application



(b) AppLock privacy setting



(c) AppLock privacy setting

Figure 3.2: AppLock protection

are protected by an application called AppLock [1]. AppLock is an ordinary Android application that protects other Android applications by a locking pattern. So, every access to an app is protected by a locking pattern. The thief is under the illusion that there is some data on the device, and it is protected. This step is necessary to convince the thief that the owner's account is being used with some defense mechanism installed. Figure 3.2 shows the AppLock functionality. This simulates the owner's account, while it frustrates the thief and tricks him/her into performing a factory reset.

It should be mentioned that the mechanism of simulating the owner's user data is independent of the concept of a honeypot anti-theft account. An alternative would be to define some plausible but fake data that is presented whenever applications are opened in the honeypot account.

In the case of a theft, all interactions will inevitably be performed in the honeypot account. In this case, the device should be modified in a way such that it executes anti-theft functionality in the background. It is open to the users and deployers of our techniques to customize the functionality of the anti-theft application. Possible functionalities for an anti-theft application here would be the collection of information of the device and the thief, remote backup functionalities as well as other remote administration functionalities that can be performed hidden from the thief. In our scenario, we implemented the anti-theft application as an Android applica-

[1] https://play.google.com/store/apps/details?id=com.domobile.applock

tion that silently tracks and logs the device location.

3.3.2 Instrumentations

When a device is stolen, chances are high that a factory reset is performed. This is usually done to make the device more usable, and to destroy any evidence of theft. A device can either be factory reset via the Android menu or by pressing a special key combination during the boot procedure. In order to save the owner's data and keep track of the thief, the factory reset is faked in both cases. This means that the device shows a realistic simulation of a factory reset and presents an empty account after the fake factory reset. The thief is in the illusion that all potential tracking mechanisms are removed and the device can now be use in an ordinary manner. Use of the device now inevitably stores the thief's personal information, which will be forwarded by the installed anti-theft application to the device owner.

We implemented the simulation of a factory reset for both mechanisms by instrumenting the Android source code. For preventing the factory reset from the Android settings menu, we instrumented the RecoverySystem[2] and RecoverySystemService[3] classes in such a way that when a user triggers the factory reset, the device shows the default factory reset animation for a realistic amount of time. We instrument the *rebootWipeUserData* function in RecoverySystem.java[2] file. In this function, a call to the *bootCommand* function is made. One of the arguments of the *bootCommand* function, specifies the intent of operation e.g., *–wipe_data* is used to wipe the data partition. Technically, when a factory reset is triggered from the honeypot account, we substitute the argument *–wipe_data* by *–wipe_cache*. This prevents the removal of the user data. Wiping the cache only removes the temporary saved files, e.g., temporary browser files. Thus, it does not affect the device owner in a negative way. Additionally for preventing data loss in case of a factory reset that was triggered during the boot process, we instrumented the recovery system[4].

During the reset, we programmatically remove the AppLock application. We instrument the *setupOrClearBcb* function in the RecoverySystemService.java file to achieve this. Since the honeypot account does not have any data, an empty account is presented to the user, that is functionally equal to a factory reset phone.

[2] /frameworks/base/core/java/android/os/RecoverySystem.java
[3] /frameworks/base/services/core/java/com/android/server/RecoverySystemService.java
[4] /bootable/recovery/device.cpp, /bootable/recovery/recovery.cpp

When a thief has logged into the honeypot account and potentially "factory reset" the device (which was simulated by our instrumentation), tracking information should be collected and forwarded to the device owner. In our approach, this is done by an anti-theft application. It is obvious that a thief should not notice that an anti-theft application is gathering information or even notice that it is installed. For this reason, in the honeypot account, the used anti-theft application is hidden from the list of installed applications in the settings menu as well as in the Android launcher. In Android there exist places where users can list the installed applications e.g., the Android Launcher and the settings menu in the category "Apps". We have instrumented these[5] such that the installed anti-theft application is hidden from a potential thief, and there are no possible traces of any installed anti-theft application anymore. For example, in the ManageApplications.java[5] file, we instrument the *onRebuildComplete* function such that the anti-theft application is removed from the list of displayed applications.

3.4 Evaluation

3.4.1 Evaluation Criteria

For evaluating our approach we determined a set of evaluation criteria that each determines the quality of one central aspect of our approach. We identified the following set of evaluation criteria (EvCrit) that together verify our approach:

EvCrit 1 - Simulating Factory Reset The first advantage of our approach is that it prevents a thief from performing a factory reset on the stolen device. This has two major benefits: First, it prevents the highly valuable personal user data from being deleted. Second, it prevents an installed anti-theft application from being uninstalled. During the evaluation we encouraged the participants to factory reset the device by every way they know. Each time after a participant finished the study, we checked whether any of the device owner's data had been deleted (e.g. by the factory reset).

EvCrit 2 - Successfully Executing the Anti-theft Application The second advantage of our approach over every other approach is that it enables the device owner

[5] /packages/apps/Launcher2/src/com/android/launcher2/AllAppsList.java, /packages/apps/Settings/src/com/android/settings/applications/ManageApplications.java

to execute an anti-theft application while the device is used by the thief. For each participant we checked whether an installed anti-theft solution was successfully executed while the participant interacted with the device and even after a "fake" factory reset.

EvCrit 3 - Indistinguishability from an Uninstrumented Device A central property of our approach is that a thief should never notice that he/she is interacting with an instrumented device. More precise: our device should be indistinguishable from a regular stock device. For this reason we checked for both of our instrumentations whether any of them was detected by the participants. This implies the following sub-evaluation criteria:

EvCrit 3a - Hiding the Faking of the Factory Reset As mentioned, during the study we motivated the participants to perform a factory reset. For every participant that performed the factory reset, we checked whether he/she was convinced that the factory reset was actually performed or experienced any irregularities (hints on the faking of factory reset).

EvCrit 3b - Hiding the Anti-theft Application, Running in the Background While a potential thief interacts with the device it is crucial that there are no traces of running anti-theft applications. For this reason we motivated the participants to note every protection mechanism installed on the device. We asked them whether the device can be used by a thief after a factory reset (implicitly asking for the presence of an installed anti-theft application) and motivated every participant to note every observed irregularity. As an evaluation for this subcriteria we inspect the number of participants that expressed by any means the presence of an installed anti-theft application.

3.4.2 Evaluation Procedure

We performed the evaluation in the form of an empirical user study. In this user study we gave each participant a Nexus 6P device that was instrumented by the implementation of ThiefTrap. As in production, our approach would be combined with any authentication mechanism for account switching, we could evaluate our approach on the main account of the device without loss of validity. Additionally, an anti-theft tracking application was installed on the device that continuously tracked the device location. Together with this smartphone, we handed out a ques-

tionnaire that asked several questions about the user's opinion of the phone. The participants were given 60 minutes time to complete the questionnaire.

In total we evaluated the answers of 35 participants. All of the participants were either students or university graduates. The majority of the participants were Master students, while also some PhD students and Bachelor students participated. While the participants studied various disciplines, the biggest group studied computer science or some computer science related studies (12 participants). One of the requirement to participate in the study was to have precise knowledge of the Android OS. We verified this via oral inquiry. The survey participant's knowledge ranged from average to expert.

3.4.3 Results

For each participant, we performed the described evaluation procedure and evaluated the answers for the mentioned questions. We inspected the device state as additional evaluation results. In the following we will discuss each of the mentioned evaluation criteria:

EvCrit 1 - Simulating Factory Reset We checked for every participant that performed the factory reset (21 out of 35) that the factory reset did not lead to a loss of any user data. The participants triggered the factory reset via the settings menu from within the operating system, as well as during the boot process via a special key combination. In all inspected cases the deletion of user data had been prevented.

EvCrit 2 - Successfully Executing the Anti-theft Application To evaluate whether the installed anti-theft application was successfully executed in the background, we implemented an anti-theft tracking application that tracked the location of the device. For every participant we checked whether the anti-theft application was executed and whether it successfully tracked the device during the study. The installed anti-theft application was successfully executed in every case independent of the user interaction. This result proves the robustness of our approach. It should be stressed that our approach is independent from the used anti-theft application. Instead of the used tracking anti-theft application every possible anti-theft application can be silently executed using our approach.

EvCrit 3 - Indistinguishability from an Uninstrumented Device As described,

we enquired for the both places where participants could potentially detect the instrumentation, whether we successfully hide the instrumentation from the users.

EvCrit 3a - Hiding the Faking of the Factory Reset For evaluating whether we could convince the participants that a factory reset was actually performed (while in reality it is just faked) we asked the following question to the participants: *"Do you think, it is possible for this person [a malicious person e.g. a thief] to completely reset the phone, in order to wipe all the owner's data, e.g. to sell the phone?"*

20 of the participants answered that they were able to perform a regular factory reset and so the device can be used by a thief. 11 participants answered that they were not able to factory reset the device as they did not know how to do it, but persons with more technical knowledge can (or could potentially) reset the device. 3 participants answered that it would not be possible to factory reset the device. When orally asked about their answers after the study, all of the participants answered that they did not know that the possibility of a factory reset exists.

One participant was not convinced of the factory reset. Due to limitations of the used AppLock application, this participant managed to access the apps before the factory reset. Thus he detected the inconsistency after the factory reset and was not convinced of the factory reset. This problem was not caused by our factory reset instrumentation but, by an implementation flaw of the used locking application. We would like to stress that the locking application is not part of our scientific contribution, but a tool we used in order to deceive a thief to believe that he/she is interacting with the real user account of the device owner. This mechanism can be replaced by every other mechanism or application that fulfills this requirement (e.g. simply filling the honeypot account with fake information).

EvCrit 3b - Hiding the Anti-theft Application, Running in the Background A core feature of our approach is to hide the existence of an installed anti-theft solution. It is necessary that a thief is not aware that an anti-theft application is running on the device (even after triggering the factory reset). For this reason, we directly asked in the questionnaire whether the study participants were able to detect any protection mechanism in the device (*"Do you think that there is a protection mechanism installed to protect the user's data? Please explain."*) In their responses to this question the 33 out of 35 asked participants answered that the only protection mechanism that is used in the device is the locking application. One participant answered that there is no protection mechanism used. Due to the mentioned limitations of the

AppLock app, it was possible for one participant to disable the AppLock app, and enter the protected applications. This participant found these applications empty and implied that there is a second protection mechanism present. First, it should be stressed that he did not imply that there is an anti-theft solution running. Second, the mechanism that simulates the owner's account is not part of our contribution and can be substituted by any other mechanism (e.g. another locking application, operating system instrumentations or simply filling the honeypot account with fake data).

Lastly, we provided space in the questionnaire to the participants where they could provide additional comments. We encouraged the participants to note every unusual observation. None of the participants noted that they observed an anti-theft application, tracking application or similar application running in the background.

To summarize the evaluation results, we found every evaluation criteria very satisfactory fulfilled. Evaluation criteria 1 and 2 were fulfilled in every case. For evaluation criteria 3a just one participant out of 35 did recognized the fake factory reset. For evaluation criteria 3b only one out of 35 participants implied that another protection mechanism is in use while he did not detected the anti-theft application. Both of these cases were caused by an implementation flaw of the used locking application. As mentioned, this locking application is not part of our contribution and can be substituted by any other protection mechanism.

3.5 Discussion

The proposed concept of an anti-theft honeypot account is novel. It provides a combination of valuable security properties that are not given by any existing approach. These security properties are the maintenance of user data (preventing user data from being deleted), the confidentiality of user data (preventing a thief with physical access to the device from reading out user data) and the accessibility of the device (enabling any remote access mechanism for the owner while the device is physically under the control of the thief). While there exist various anti-theft solutions, none of them can fulfill all of these properties.

An important benefit of the proposed approach compared to existing anti-theft solutions is that *a device instrumented with our approach is indistinguishable from an*

ordinary device. A thief can never tell whether he/she has stolen an ordinary device or a device instrumented with the anti-theft honeypot account. Studies [19] have shown that 34 % of Android devices are not protected by a locking mechanism, so there is no way for a thief to determine whether our mechanism is used or not. Also a noticeable proportion of devices are protected by the app locking mechanism that we use to protect user app data initially. So from the existence of a locking app, a thief can never imply the existence of an anti-theft honeypot account. Another aspect that plays a role in this matter is the flexibility of our approach. A locking app is just one mechanism for faking the honeypot account. An alternative for future work is the creation of fake user data. This data will then simulate the user data of the owner, while protecting his privacy. This data can be created either by the deployers of the anti-theft honeypot account, the users or both of them in cooperation.

Device theft is a serious problem with a rapidly growing number of reported cases. Studies [45] reported that in 2013 more than 3 million devices have been stolen. For this reason Google has taken steps to mitigate damage in case of a device theft. The two most important measures to mention here are the Android Device Protection mechanism [29] and anti-theft functionalities within the Android Device Manager [28]. The Android Device Protection mechanism requires that after a factory reset, a user logs into the device with the credentials of his primary Google Account. As a thief can not know these credentials, even after a factory reset, the stolen phone is of no use. The Android Device Manager can be used to track a phone and to remotely wipe. Compared with the combined usage of these two native Android tools, our approach has two advantages: First, it prevents the deletion of user data. Nowadays, a plenitude of valuable user data is stored on modern smartphones. In the vast majority of cases, the user data on such phones is hard to recover or even irreplaceable. In contrast to the mentioned Android tools, our approach can prevent the loss of this data. The second advantage is that any anti-theft application and functionality can be executed while the device is stolen. In contrast, the Android Device Manager just supports tracking and wiping functionalities.

Physical access to a device enables a number of novel attacks, so called hardware based attacks. In the context of Android smartphones prominent examples of these attacks are that by Cannon and Bradford [12] and the work of Ossmann and Osborn [63]. Cannon and Bradford used a so called white card, a special SIM card that au-

thorizes flashing of a custom ROM on a device. Among others, from such a ROM it is possible to read out user data. Also Ossmann and Osborn proposed a hardware based attack with which it is possible to read out user data. By connecting to the Micro USB connector via UART with TTL logic they could connect to an integrated debugger, which enabled them to activate the Android Debugging Bridge functionality and so gain access to the device. Relating to our work it should be mentioned that in principle such hardware attacks might also be possible on devices with our instrumentations in place. Still, it should be stressed that hardware attacks like these require expert knowledge of the used hardware technologies and an existing vulnerability in the smartphone device. It is unlikely that both of these factors apply in the average case of a stolen device and so the impact of hardware based attacks on our proposed approach is negligible.

Another factor that should be discussed in the context of hardware attacks is the confidentiality of user data saved on a SD card in the device. While the AppLocking mechanism protects the confidentiality of user data on the SD card from access within the smartphone, the SD card can also be extracted from the device and read within another device. To protect the confidentiality of user data in this scenario it is necessary to encrypt the content of the SD card. Such an encryption of the SD card is orthogonal to our approach and can be implemented as completion to this approach.

We implemented our changes into the AOSP (Android Open Source Project). Additionally, our changes are portable, generic and thus can be easily integrated with every vendor specific build.

The usability of a device instrumented with our approach may differ from the normal device in terms of logging into the real account. For devices with a fingerprint scanner, the usability is not affected because the login procedure to the real account is equal to an ordinary login. Devices without a fingerprint scanner will require users to enter a pattern in the Honeypot account. This pattern can be configured by users. It is similar to unlocking a device using a pattern or a PIN. Hence, the usability impact is negligible.

Lastly, we would like to stress the flexibility of this approach. The concept of an anti-theft honeypot account can be applied orthogonally to any existing anti-theft mechanism. The honeypot account is responsible for protecting the user data while simultaneously any anti-theft mechanism is implemented. The benefit of the

proposed concept is that in contrast to other approaches where a thief will quickly factory reset the device, in this approach it is likely that he/she will even establish an internet connection and so enabling the remote access for the installed anti-theft solution.

Part III

Privacy/Data Protection Against (Un)Intentional Application Leaks

Chapter 4

IIFA: Modular Inter-app Intent Information Flow Analysis of Android Applications

4.1 Overview

To protect sensitive information on Android, various information flow control (IFC) analyses have been developed. These analyze the (potential) flow of information in apps and report a warning if a flow from a sensitive data source to an untrusted/public data sink (like sending sensitive information to the internet) is determined. Information flow is not restricted to a single component, but occurs frequently between components of the same [53, 36] and even different apps [84]. Our study using the top 90 apps from the Google play store revealed more than 10,000 inter-component calls. Scrutinizing the flows between components therefore becomes imperative.

Android's ICC mainly leverages so-called *intents*. The major challenge in identifying IFC through intents is identifying which information flows from one component to another. Leveraging static analysis is non-trivial because the receiver and the intent data may be unknown at analysis time, being strings that might be composed at runtime.

Some tools consider intents during information flow analysis [53, 84, 9, 50] but suffer from multiple shortcomings: These approaches basically inline a synthetic "main" method that models the lifecycle of the receiving component/app into the sender of the intent. Inlining several sender and receiver apps into one huge app to analyze (e.g. 90 apps like in our evaluation study) would require a very precise analysis in order to not magnify the imprecision described in the last paragraph (e.g., context-sensitivity to match multiple senders to the same component lifecycle, or even that one sender is only problematic if it has received sensitive intent data from another app). Unfortunately, inlining several apps into one raises immediate scalability issues as even single app ICC is challenging in terms of scalability [53]. Therefore an alternative approach might be to eagerly analyze all pairs of apps. Note that at least a quadratic number of app combinations would have to be analyzed in such a scheme to include the effects of IAC to IFC. Realistically, intent communication can involve more than two apps, further aggravating the combinatorial explosion of merging-based approaches. Besides, merging itself is impractical in two dimensions: Merging APKs or other internal data structures does not scale to realistic apps in our experience, and even if it does, the complexity to analyze the merged app inflates, but as most combinations of apps do not communicate via intents the whole effort is mostly futile. Simultaneously the merging process itself may introduce spurious data flow paths, increasing analysis imprecision.

However, in order to match senders and receivers these approaches merely verify that the intent action (or similar receiver-identifying data) matches. Our evaluation shows that none of the ICC-considering approaches actually determines whether the receiver and the sender use the same type and key in the key-value communication scheme of *extra data* transmitted via intents (discussed in detail in section 4.4.1). Thus, either some or even all receivers in the inlined lifecycle might not be eligible to read the transmitted data, which can thus result in many spuriously reported data leaks. A more accurate matching could probably be added to these approaches, but only if there is merely one sender and one receiver statement in any given pair of communicating components. In case of multiple sender and/or receiver statements one matching pair might still induce a quadratic number of spurious flows that would require more involved constraint solver technology to remove.

Finally, related work suffers from requiring access to source code (or even source code annotations) [43, 42, 8, 61], lacking support for string analysis [5, 53], and lack-

ing support for certain sink functions [5, 53, 43, 50]. All of these drawbacks lead to insufficient precision and soundness issues when these analyses are applied to real-world apps.

4.2 Our Contributions

In this work we propose a novel information-flow analysis for IAC (and ICC) based on an intent-flow pre-analysis that evades combinatorial explosion of analyzing all potential communication partners, while precisely matching type and key information of intent data. Our approach can predict which combinations of apps communicate by separating information flow analysis within app components and matching of communication partners. In a first step we create a database of summary information about senders of intents, their characteristics including types and keys, and outbound intent data, as well as apps registered to receive certain implicit intents. This information can then be matched in a subsequent step to identify potential communication partners. Further we propose a novel matching algorithm, based on a baseline IFC analysis providing potential intra-component flows (including a program slices of the receiver's key value) for all potential intent receivers. We leverage senders' outbound intent data as input to the information flows identified in respective receivers, which eliminates the need for inlining or merging apps and thus combinatorial explosion, as only summaries of actual communication partners are subsumed. In case multiple apps are involved in intent communication our approach performs a light-weight fixed point iteration through the DB information. Remember that our tool is not a stand-alone IFC analysis tool. Rather, IIFA leverages flows and slices generated by other IFC analyzers. As these tools are already heavily engineered for the intra-app case, we concentrated on the peculiarities of intent communication and evasion of inlining and combinatorial explosion.

As a noteworthy novelty, our approach is modular and thus compositional with respect to app installation. Whenever a new (version of an) app is available for analysis, the database is updated (in case of new version) or extended (new app) to include the intents broadcast or received by this app. Only the new app has to be (re-)analyzed, as well as combinations with flows identified in potential receivers. We compute precise communication paths between components, handling complex control flows such as in callback methods (see section 4.4.2). As intent targets are

specified via a string parameter, our approach can resolve common string manipulations, which improves our precision significantly for regular (non-obfuscated) apps. As a minor contribution we took great effort to handle the full spectrum of intent communication, supporting explicit and implicit intents as well as dynamically created intent receivers. All previous approaches miss at least one of these features, leading to unsoundness and imprecision. Our analysis is fully automated and does not require the source code of the app under analysis. We aim to answer the following research questions:

- *Does our pre-analysis approach negatively impact the precision or soundness of the results with respect to state-of-the-art analyses?*
- *Does our approach scale to a realistic corpus of real-world apps?*
- *How do common real-world apps communicate through IAC?*

We implemented our approach as a tool called *IIFA* and evaluated it on DroidBench, the IccTA extension of DroidBench, ICC Bench, our own benchmarks evaluating key and type matching of intent extra data, and a large set of apps from the Google Playstore. We compared our results with multiple related analysis tools. Our tool (combined with an external baseline intra-component IFC analysis) achieves perfect precision and soundness on all benchmark sets, being more than on par with related IFC tools that consider intent communication. Additionally, we demonstrate that IIFA can improve the IAC precision of other base IFC analyses with experiments, and assess the scalability of IIFA, applying it to the 90 most downloaded Playstore apps. Our experiments demonstrate that due to its compositionality IIFA's execution time scales well even to a large corpus of real-world apps. In summary, we provide the following contributions:

- *Compositional DB-backed Analysis.* We propose a modular pre-analysis approach for intent communication, in particular for analyzing inter-app communication, based on summaries for all app components containing intent senders, receivers, and the exact intent characteristics including types and keys of data transmission. To that end, we model all publicly known intent-based communication schemes precisely.
- *Novel Matching Algorithm.* We present a novel algorithm which matches intent senders with intent receivers based on these summaries and even detects flow

through more than two components via a lightweight fixed-point iteration. Our approach subsumes transmitted data into the receiver's intra-component information flows (provided by a baseline IFC analysis) to report potential dangerous inter-component and -app information flows. This matching requires no eager pairwise analysis but only investigates potential communication partners. Thus each app is only analyzed once by IIFA and potentially as well by an intra-app baseline IFC analysis.

- *Evaluation of IIFA.* We implemented our analysis (IIFA) and evaluated it on multiple large-scale datasets. The evaluation shows that our pre-analysis approach does not negatively impact precision and recall with respect to the most relevant previous work on benchmarks, including a novel suite assessing the correct matching of key and type of intent extra data. We demonstrate that we can effectively evade combinatorial explosion, analyzing ICC/IAC information flows of the top 90 real-world apps in approximately 2.2h (excluding the baseline IFC analysis) and identifying 62 potentially dangerous information flows through ICC. Finally we performed a study regarding the use of ICC/IAC in Android and present our results outlining IAC patterns in realistic applications.

```
1 TelephonyManager tel = (TelephonyManager)
      getSystemService(TELEPHONY_SERVICE);
2 String imei = tel.getDeviceId(); // source
3 Intent i = new Intent("CUSTOM_INTENT.ACTION");
4 i.putExtra("data", imei);
5 startActivity(i); // sink
```

Listing 4.1: App A - Sender: OutFlowActivity

```
1 <intent-filter>
2   <action android:name="CUSTOM_INTENT.ACTION"/>
3   <category android:name = "android.intent.category.DEFAULT" />
4 </intent-filter>
```

Listing 4.2: App B - Receiver: AndroidManifest.xml

4.3 A Motivating Example

In this section we will describe an example workflow of intents. We will then discuss the inefficiency of the current state-of-the-art analysis tools. In Listing 4.1, *App A* initiates IAC in the *OutFlowActivity* class. An implicit intent *i* is created (line 3) with the intent action *"CUSTOM_INTENT.ACTION"*. The *putExtra* method (line 4) associates additional data from the variable *imei* with the key *"data"* in this intent. The device id stored in variable *imei* (lines 1 and 2) is sensitive data, as the device can be uniquely identified by this number. The intent is finally triggered via a *startActivity* call such that it can be received by registered receivers. On the receiver side the intent extra data can be extracted by the receiver and (ab-)used in any way permissible to that app.

Listing 4.2 and 4.3 show the code snippets of an example receiver for the above intent. In the manifest file (Listing 4.2) the receiver declares its capability to support intent filter (*"CUSTOM_INTENT.ACTION"*). Listing 4.3 extracts the received intent

```
1 Intent i = getIntent();
2 String imei = i.getStringExtra("data");
3 smsManager.sendTextMessage("1234567890", null, imei, null, null);
      // sink
```

Listing 4.3: App B - Receiver: InFlowActivity

data corresponding to the key “data" via a *getStringExtra* method call. This data flows to a data sink (line 3) where it is being leaked off the device. Since the received data is a sensitive information with respect to App A, analysis tools should report this as a potential data leak. Observe that these two apps must be related by an analysis in order to identify the leak and that the precision for determining intent data crucially influences the precision to determine related apps and which data is transmitted.

Current analysis tools for ICC [53]/IAC [84] only match the identifying string (like the intent action in Listing 4.1- 4.3) and ignore the key “data” or the type (in our case *String*, see line 2 of the receiver), which must also match for data to be transmitted. If multiple receivers are present in one component then adding checks for these conditions is non-trivial as these approaches basically inline the receiver into the sender, thus the same intent would be used for all receivers even if the key or type of the *getExtra*-method differs, resulting in spurious reported information flows. Simply merging/inlining all apps installed on a device is prohibitively expensive, and may result in impossible flows created as a result of the merging process, thus in practice these approaches may resort to eagerly inlining pairs, triples, ... of communicating apps, leading to combinatorial explosion of analysis targets. Furthermore, in a realistic scenario apps may be installed on a device at any time. Thus, whenever a new app (version) arrives, these tools need to join apps again.

To overcome these limitations, our analysis is designed in a modular way: It remembers summaries of the intent characteristics (including key and type) of each app in a database and applies this knowledge to (intra-app) information flows determined in receiving apps. Due to the database our analysis can recursively resolve dependences when more than two apps are involved in intent communication.

4.4 Methodology

The fundamental problem of intent analysis for static analysis is the dynamic nature of intents. Static IFC analyses generally leverage dataflow analyses like backwards slicing to determine whether sensitive information (e.g., a device id) may flow at a sink (e.g., internet). However, if a slice contains statements where data is extracted from a received intent, it cannot determine the data’s sensitivity without detailed knowledge on possible senders and their semantics.

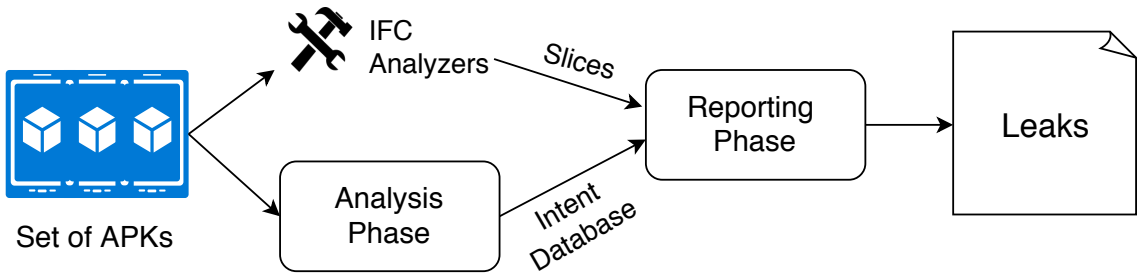


Figure 4.1: Analysis Framework

Figure 4.1 presents the major building blocks of our analysis framework. In the *analysis phase* a set of APKs under inspection (e.g., all apps installed on a device) is processed and the extracted information stored into a SQL database named *IntentDB*. We collect two sets of information, app-specific information, i.e., package and class name, and registered intent filters to receive implicit intents, as well as intent sender-specific information, i.e., information required to identify potential receiver(s), key, type, and the actual data being sent.

The database is fed into the *reporting phase* together with the receiving app's information flows from a baseline (intra-component) IFC analyzer. If a flow originates at a *getXXXExtra* method[1], we consider the respective sender's outbound data as the actual data source to that flow. Remember that data can only successfully be transmitted via *put/getExtra* methods if the key parameters of both methods match and the signatures of the *put* and *get* methods correspond (e.g. the value's type of the *put* method equals the return type of the *get* method). Thus we determine all potential senders of this intent based on matching the target component or intent action. For each of these senders we extract the *key*, *value*, and *put* signature (see section 4.4.1) from database. If the key and the put signature match this *getXXXExtra* method invocation[2], we determine the sensitivity of the transmitted value based on a categorization of sources. If the value is considered sensitive, we report a potential information flow violation.

As an example, consider Listings 4.1 and 4.3 again: The *sendTextMessage* (line 3) receives data from the *getStringExtra* method (line 2). Therefore we scan our database for potential intent senders of App B's received intent (line 1): App A sends an implicit intent with matching intent action ("*CUSTOM_INTENT.ACTION*", line 3 of

[1] *getXXXExtra* methods retrieve type-specific data from a received intent that has been added through the corresponding *putXXXExtra* method.

[2] The *getXXXExtra*'s key is determined via backward slicing

Table 4.1: Example database for a class that can receive as well as send intents

Package Name	Class Name	Intent Filter	Target Component	Intent Action	Key	Value	Put Signature
org.telegram.messenger	Firebase InstanceId Service	com.google.firebase.INSTANCE_ID_EVENT	null	com.google.android.gcm.intent.SEND	"google.to"	String url = "google.com/iid"	putExtra (String, String)

Listing 4.1). The signature of the *putExtra* method (line 4) has a *String* parameter, which matches the return type of the *getStringExtra* method of the receiver, and the keys of the sender and receiver ("data") match. Thus, the source of the transmitted value (i.e. the IMEI of the device) is considered the information source of the flow to the SMS transmission in App B. As the IMEI is sensitive information, an information flow violation is reported.

Structure of *IntentDB*: Table 4.1 shows an example entry (from the Telegram messenger app) of the database. As apps consist of several classes, this table has potentially multiple entries for the same app. All entries belonging to one app can be identified by the unique package name. Similarly, each class can send out several intents and hence for each intent sent we will list a separate entry (package name & class name are the same). The column "Put Signature" is considered for mapping the *put* method to the corresponding *getXXXExtra* method at the time of intent resolution. Depending on the non-empty fields, an entry in the database represents an intent receiver and/or sender. If the *Intent Filter* field is set, the app may receive intents. If either the *Target Component* or the *Intent Action* field is set, it acts as an intent sender.

4.4.1 Analysis Phase

Figure 4.2 depicts the workflow of the analysis phase. In the sequel, we describe the details of each component:

Apktool

A set of APKs is processed by *Apktool* [1], which extracts and decodes the resources of an APK (e.g., *manifest.xml*). It decodes the Dalvik bytecode file (*classes.dex*) of the

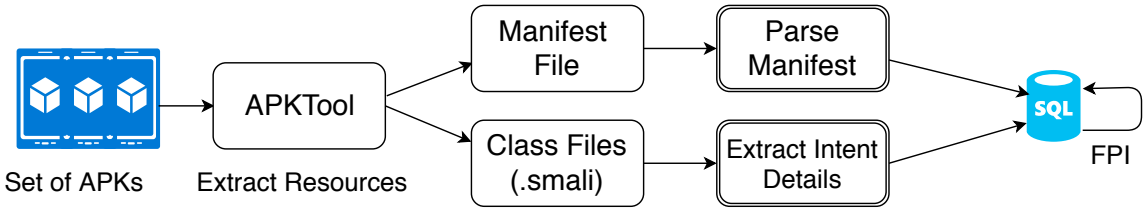


Figure 4.2: Analysis Phase, FPI stands for fixed point iteration

APK to more comprehensible Smali class files [23].

Manifest Parser

Parsing the manifest file extracts various app details (first set of information), i.e., *package and class name*, as well as *supported intent filters*. This information is mapped to the first three columns of the table and identifies potential receivers of an intent. Even though intent receivers are typically registered in the manifest file, the *registerReceiver* method can register an intent receiver at runtime. In our experiment with 90 apps, we find 433 dynamically registered receivers ($\approx 5\%$ of all intent receivers). We scan class files for dynamically registered receivers and store them in IntentDB.

Dynamic Intent Data Extraction

In this module, we scan each class file for methods that initiate an intent (sender methods), e.g., *startActivity*. The *Android documentation* [25] defines 25 such methods including 12 variants of *startActivity*, 11 variants of *broadcast*, *startService* and *bindService*.

Identifying Target Component/Intent Action For every sender method we compute its backward slice and trace until we find the corresponding intent initialization(s). The goal is to identify its target component (for an explicit intent) or intent action (implicit intent). The intent type depends on the intent's constructor but can be altered using the *explicit-transformation* methods *makeMainActivity*, *makeRestartActivityTask*, *setClass*, *setClassName*, *setComponent*, *setPackage* or *setSelector*, which can also *change* the target component after the fact. We analyze these cases to extract the actual target: In the case of an explicit intent, we identify the name of the target component. For an implicit intent, we extract the intent action. Any app

defining this intent action as supported intent filter (dynamically or in its manifest file) is a potential receiver of this intent. Unfortunately, one cannot always statically determine intent details (e.g., intent action) as they may be influenced by runtime information, which is a general limitation of static analysis. We conservatively approximate such situations, i.e., may include several potential intent actions into the database. Future work may rule out non-matching substrings of potential target name/action strings similar to reflection analyis [37].

Identifying Key-Value Pairs There are several methods to associate extra data with an intent, generally leveraging *key-value* pair schemes. Senders register a value specifying the key, e.g., *Intent.putExtra("Test-Key", "Test-Value")* will register the string *"Test-Value"* as data for the key *"Test-Key"*, which can be extracted by a corresponding receiver using the *Intent.getExtra("Test-Key")* method. Trying to receive a key with a non-matching data type results in no value being transmitted. Therefore precise analysis mandates a correct matching of *get* and *put* methods. Unlike related work [53, 84] we handle the respective *put/get* method pairs for all basic data types and store the precise signature of any *put* method in *IntentDB* to consider matching types and keys when resolving values received by *getXXXExtra* methods at intent receivers.

Fixed Point Iteration

Intent communication may involve more than two apps/components. In our experiments with 90 apps, we find 54 cases where more than two components were involved in a transitive information flow. In such a case, *IntentDB* contains a *getXXXExtra* method in the column *Value*. For example, in Listings 4.4, app A is sending the device id (secret data) to app B. App B forwards this data to app C, and finally app C leaks it via an SMS. The first 3 rows of Table 4.2 show the table *IntentDB* prior to fixed point iteration. To resolve transitive flows through multiple components we iterate in a fixed point iteration through the entries of *IntentDB* for which *Value* contains a *getXXXExtra* method. The *com.appB* entry in Table 4.2 is such an example where data from a received intent is being sent out via another intent. In order to identify the received data, we determine all apps from which this component could receive the intent on which *getXXXExtra* is invoked. In our example *com.appB* receives from *com.appA*. Finally we match the corresponding *key-value*

```
1 // APP A (OutFlowActivity)
2 TelephonyManager tel = (TelephonyManager)
      getSystemService(TELEPHONY_SERVICE);
3 String imei = tel.getDeviceId(); // source
4 Intent i = new Intent("action_test");
5 i.putExtra("data", imei);
6 startActivity(i); // sink
7
8 // APP B (Intermediate Activity) -- Capable of receiving
      "action_test"
9 Intent i = getIntent();
10 String imei = i.getStringExtra("data");
11 Intent newIntent = new Intent("action_test2");
12 newIntent.putExtra("secret", imei);
13 startActivity(newIntent);
14
15 //APP C (InFlow Activity) -- Capable of receiving "action_test2"
16 Intent i = getIntent();
17 String imei = i.getStringExtra("secret");
18 smsManager.sendTextMessage("1234567890", null, imei, null, null);
      // sink
```

Listing 4.4: Sender: OutFlowActivity

Table 4.2: *IntentDB* for Listing 4.4. Fixed point iteration adds the last row

Pckg. Name	Class Name	Intent Filter	Target Component	Intent Action	Key	Value	Put Signature
com.appA	OutFlow Activity	null	null	action_test	"data"	*Device ID*	putExtra (String, String)
com.appB	Interm. Activity	action_test	null	action_test2	"secret"	getStringExtra("data")	putExtra (String, String)
com.appC	InFlow Activity	action_test2	null	null	null	null	null
com.appB	Interm. Activity	action_test	null	action_test2	"secret"	*Device ID*	putExtra (String, String)

pair through their *get-put* signatures and create a new entry, replacing the original source (*getXXXExtra* method) by the transmitted value. The created entry for our example is shown in gray in Table 4.2. To accommodate for modular analysis and thus potential new compatible senders, we retain the old database entry (row 2). The reporting phase described in the next section now matches the added row with the intent received in App C to reveal the transitive information flow of sensitive data to the SMS sink.

4.4.2 Reporting Phase

In the reporting phase, we process information flows obtained by a baseline IFC analyzer together with the *IntentDB* from the analysis phase. For ICC/IAC we are only interested in flows with sources that are potential intent receivers, i.e., a

```
1 public class OutFlowActivity extends Activity{
2   protected void onCreate(Bundle savedInstanceState) { // ...
3    TelephonyManager tel = (TelephonyManager)
        getSystemService(TELEPHONY_SERVICE);
4    String imei = tel.getDeviceId(); // source
5    Intent i = new Intent(this, InFlowActivity.class);
6    i.putExtra("data", imei);
7    startActivityForResult(i, 1);
8   }
9   protected void onActivityResult(int requCd, int resCd, Intent
       data) {
10     String imei = data.getStringExtra("data");
11     smsManager.sendTextMessage("1234567890", null, imei, null,
          null); // sink
12 }}
```

Listing 4.5: Sender: OutFlowActivity

getXXXExtra method (together with its key and signature). For every *getXXXExtra* method in a reported information flow, we extract all potential senders to this receiver from *IntentDB*, i.e., apps that use an intent with a matching target component or a matching intent action. Finally, we match *get-put* method pairs and *keys* to determine senders that actually send data to this receiver and report it as a (potential) leak if the transmitted data stems from a sensitive source[3].

For example, data flows from the *getStringExtra* method of the intent received on line 16 to the data sink *sendTextMessage* in App C. Our analysis thus matches any sender of the intent action *action_test2* and finds two rows in IntentDB (Table 4.2). We check whether any of those uses the key *secret*, which both of them do. Then we match the signature of *getStringExtra* with the sender's Put Signature, where again both match. Finally, we verify if one of the potentially transmitted values (*Device ID, getStringExtra("data")*) is sensitive, thus reporting the former as an illicit information flow.

Handling of *startActivityForResult* and *bindService*

startActivityForResult is a special case of intent communication illustrated via a code snippet of an activity in Listing 4.5 (adapted from [18]). *OutFlowActivity* (line 5) creates an explicit intent with *InFlowActivity* as the target component. This intent is

[3]We utilize the categorization of sources and sinks from R-Droid [7]

```
1 public class InFlowActivity extends Activity {
2   protected void onCreate(Bundle savedInstanceState) { // ...
3    Intent i = getIntent();
4    setResult(1, i);
5    finish();
6 }}
```

Listing 4.6: Receiver: InFlowActivity

provided extra data *imei* (line 6), containing the actual *IMEI* of the device (lines 3, 4). *startActivityForResult* triggers this intent (line 7) with a second argument that is a *request code* identifying this request. Listing 4.6 contains the code snippet for the activity *InFlowActivity*, receiving this intent (line 3). The *setResult* method (line 4) returns the received intent with the same *request code*. Upon successful creation of *InFlowActivity* control returns to the *onActivityResult* (line 9 of listing 4.5) method of *OutFlowActivity*. The third parameter (*data*) of this method corresponds to the intent returned via the *setResult* method of *InFlowActivity*. This intent, originally sent by *OutFlowActivity*, still contains the secret *IMEI* of the device. The *IMEI* is extracted (line 10) and leaked (line 11) via a text message. Thus data flows from the sender to the receiver and back, as modeled in our information flow analysis.

Similarly, after a Service has been successfully bound via *bindService*, control will return to an *onServiceConnected* method (of an object designated as the second parameter of the original *bindService* call.) This method is being passed an argument from the service intent receiver, and thus data can be returned to the intent sender. In order to identify such flows soundly and precisely our analysis models the data flow according to these patterns at both sides of the communication.

4.4.3 Domain Knowledge for Java String Class and List Analysis

As the ability to precisely determine intent senders and receivers depends significantly on the ability to identify the String values of target components or intent actions, we enrich IIFA with domain knowledge on the Java String class. IIFA understands the Smali signature of String methods and applies partial evaluation in order to recover strings created by concatenation, substring, and other String manipulation methods. Concretely, it extracts parameters, applies the respective functionality and returns the resulting string. More contrived examples like converting

a string to an array of chars (to be manipulated) are beyond the scope of our tool as we are currently not targeting obfuscated code. Due to our modular design we could also add more expensive analyses like SMT-solvers that handle more cases. However, there are always undecidable cases like encrypted strings or dynamic input.

Similarly, we encode domain knowledge on the API of *LinkedList* to be able to extract list entries from a given index they were stored in. Again, a more precise model of Lists improves analysis precision but in general this problem is undecidable. Other collections could be modeled analogously, which we are planning as future work.

4.5 Evaluation

We empirically evaluated our tool, IIFA, in two steps:

- *Comparative evaluation on benchmark sets.* We applied IIFA to four standard evaluation sets for intent communication comprising 48 test cases with ground truth results for each test. We compared the precision and soundness of IIFA to six state of the art tools that support intent analysis.
- *Evaluation on real-world apps from the Google Playstore.* We applied IIFA to the 90 most popular apps from the Google Playstore in order to evaluate its scalability on real-world apps.

All experiments were performed on a MacBook Pro with a 2,9 GHz Intel Core i7 processor and 16 GB DDR3 RAM and MacOS High Sierra 10.13.1 installed. We used a version 1.8 JVM with 4 GB maximum heap size.

4.5.1 Precision and Soundness of IIFA

Benchmark evaluation datasets

Remember that IIFA is not a stand-alone tool. Therefore its intention cannot be to replace any of the related works that analyze intra-component information flows. Rather we are propagating the idea that ICC/IAC analysis needs to be done in a

pre-analysis, and our experiments in this section are to show that this design decision does not negatively impact the precision or soundness of the analysis results. In order to evaluate the precision and soundness we use four separate benchmark sets and compare the results of IIFA (combined with a basic intra-component information flow analysis) to related approaches that aim at analyzing both intra-component and ICC/IAC information flows simultaneously. As IIFA is not a standalone IFC tool but a pre-analysis modeling the information flows through intents, we restrict our experiments to the subset of the benchmarks that evaluate precision and soundness of intent-based communication:

- The intent-related cases of the original DroidBench test suite [18] **(14 test cases)**
- The extension proposed by IccTA [53] **(18 test cases)**
- ICC-Bench, proposed by Wei et al. [84] **(9 test cases)**
- Our extension[4], which evaluates correct matching of types and keys for data exchanged via intent extra data **(7 test cases)**

Note that the mentioned benchmark sets include several advanced usage scenarios of intents. An example of these scenarios is the usage of callback methods that are triggered after an event has been delivered to its target, which requires information tracking at both sender and receiver sides (see Section 4.4.2). Another challenge is string manipulation, e.g., of keys for intent extra data. Finally one case passes an intent with sensitive data through multiple components before finally leaking the stored data. The authors of each benchmark set provide ground truth for each test case, which we use to measure precision and soundness. In the following we will give a short description of each of the used datasets.

Original, intent-related DroidBench test cases DroidBench is a set of Android apps, proposed by Arzt et al. [18]. "[It] contains test cases for interesting static-analysis problems (field sensitivity, object sensitivity, tradeoffs in access-path lengths etc.) as well as for Android-specific challenges like correctly modeling an app's lifecycle, adequately handling asynchronous callbacks and interacting with the UI [18]." As our approach is focused on intents, we filtered out the test cases that do not include the usage of intents and applied IIFA on all of the remaining 14 test cases.

[4]https://github.com/mig40000/ICC-Benchmark

Table 4.3: Summary of Tool Results for Micro-Benchmarks

ICC Comparison							
IccTA Extension + ICC-Bench + Attribute-Mismatch-ICC-Benchmark (34 test cases)							
Precision, Recall and F1-measure	FlowDroid	AppScan	DidFail	DIALDroid	Amandroid	IccTA	Our Tool
Precision $p = ✓/(✓ + ⋆)$	25%	16.7%	75%	71%	62%	80%	100%
Recall $r = ✓/(✓ + ○)$	80%	62.5%	24%	80%	60%	96%	100%
F_1-measure $2pr/(p+r)$	0.41	0.26	0.36	0.75	0.61	0.87	1
IAC Comparison							
DroidBench IAC + Attribute-Mismatch-ICC-Benchmark (10 test cases)							
Precision $p = ✓/(✓ + ⋆)$	0%	0%	63%	73%	52%	0%	100%
Recall $r = ✓/(✓ + ○)$	0%	0%	21%	56%	76%	0%	100%
F_1-measure $2pr/(p+r)$	0	0	0.31	0.63	0.43	0	1

IccTA extension of DroidBench Li et al. [53] proposed IccTA, an extension of DroidBench that includes further test cases for inter-component communication. This set covers additional methods for inter-component communication. Four of these test cases (*startActivity 4 –7*) do not include any actual leaks and are included to detect false positives. Additionally, each test case includes an unreachable component that contains a leak. These components are used to verify that the analysis tools perform a proper reachability analysis. Four out of the 22 test cases in this benchmark set include inter-component methods that are not intent-related. As our tool is focused on intents we evaluated IIFA on the remaining 18 test cases.

ICC-Bench Wei et al. [84] developed ICC-Bench, a benchmark set for inter-component communication. This benchmark set has nine test cases including six implicit intent cases. In the remaining three cases, intents are either explicitly constructed or dynamically transformed to explicit intents by specifying a target component after intent construction.

Our Extension We developed Attribute-Mismatch ICC-Bench, a benchmark to test the key and/or type mismatch (see section 4.4.1) in ICC/IAC. This benchmark contains seven test cases, six ICC and one IAC. This benchmark focuses on validating the creation of invalid communication paths due to a mismatching key and/or type of intent extra data during ICC/IAC resolution. This is a feature that was neglected by previous benchmark suites but has great impact on the precision of the communication results.

Comparative evaluation

Based on true positives (tp), false positives (fp), and false negatives (fn) we use the following metrics to compare the performance of IIFA with the related tools:

Precision	**Recall**	F_1**-measure**
$p = \frac{tp}{tp+fp}$	$r = \frac{tp}{tp+fn}$	$\frac{2pr}{p+r}$

We applied IIFA to the original DroidBench benchmark set, where 14 test cases are relevant for intent communication. On these benchmarks IIFA achieved perfect precision and recall ratios. We further applied IIFA to the IccTA extension of Droidbench [18], ICC-Bench [84], and Attribute-Mismatch-ICC-Benchmark, and compared the results to the six most prominent tools for Android intent information flow analysis: FlowDroid [5] and AppScan [43] are limited to ICC, IccTA [53] and AmanDroid [84] require an additional tool to support IAC, DidFail [50] and DIALDroid [9] come with their own inter-app analysis. Table 4.3 summarizes the metrics for the results of the different tools on these benchmark sets[5] (results for related work on previous benchmarks taken from Li et al. [53]). In the sequel, we compare the results of FlowDroid, AppScan, DidFail, DIALDroid, Amandroid, IccTa and IIFA.

FlowDroid FlowDroid [5] was designed to analyze information flows in isolated Android components. Li et al. [53] added an additional analysis step that combines the paths between components. As a precise computation of inter-component paths is not possible, these need to be overapproximated, which leads to spurious paths and thus false positives. These circumstances lead to a low precision (25%) with a medium recall (80.0%). As FlowDroid does not handle certain callback methods, it returns false negatives in three of the *bindService* test cases.

AppScan AppScan only supports intra-app inter-component communication analysis but does not natively support other inter-component flows. In order to overcome these restrictions, its flows were combined by [53], resulting in a low precision (16.7%) and a medium recall (62.5%). Additionally, AppScan cannot handle the *startActivityForResult* method, leading to false negatives in the test cases *startActivityForResult 2* to *4*.

DidFail DidFail is only able to analyze Activities and thus cannot detect any

[5] A detailed comparison on each test case is present in Table 1 of Appendix A.

leaks in the three other types of Android components (Services, Broadcast Receivers and Content Providers). Additionally, it ignores leaks that include explicit intents. These circumstances lead to a very low recall rate of 24%. Finally, DidFail includes imprecision in that it does not consider mime types and data. This leads to false positives in the test cases *startActivity 4 and 5*.

DIALDroid DIALDroid is designed specifically to detect ICC and IAC flows only and it ignores pure intra-component flows. It could not resolve a mismatching key and/or type as well as test cases involving non-trivial string manipulation. Additionally, it aborted several times analyzing implicit intents, especially if more than two apps were involved in the information flow. This restricts the utility of this tool as implicit intents are more prone to unintended information flows.

Amandroid Amandroid has a mediocre recall rate of 60% due to imprecision in complex sender functions (section 4.4.2), imprecision in the lifecycle model of Services (*startService2*) and string manipulation (*DynRegister2*) and lacking support for Content Providers. In addition, Amandroid cannot handle implicit flows [65]. There are multiple reasons for the false negatives. First, Amandroid does not map the arguments passed to the *setResult* method of the intent-receiving component to the arguments of the callback method *onActivityResult* in the sender component. The failure of the *startService2* test case indicates some imprecision in the lifecycle model of Services. Further, Amandroid cannot properly handle calls to the *bindService* method. Finally, Amandroid fails to track string manipulation operations that lead to a false negative in the *DynRegister2* test case.

IccTA IccTA displays above average precision (80%) and a very good recall ratio (96%). It fails on test cases that include non-trivial string manipulations, e.g., *DynRegister2* and *startActivity7*. IccTA is reported to suffer from scalability issues, the experiments reported in Li et al. [53] and subsequent publications [9, 66, 64] are restricted to ICC and to the previously known micro-benchmark suites; no evaluation on real-world apps has been reported.

Matching the key and/or type of intent extra data is not reported in Li et al. [53], DidFail [50], or AmanDroid [84]. They merely create a lifecycle method that connects the sender of an intent with (the respective) receiver(s), thus creating a data flow between these components. In our experiments both IccTA and Amandroid failed to detect a key and/or type mismatch during ICC (see section 4.4.1) due to missing checks of these constraints, resulting in significant precision loss with re-

spect to previous benchmark suites alone (details can be found in Appendix A).

Our approach: IIFA To evaluate that IIFA's pre-analysis approach does not negatively impact the precision and soundness we applied it to the same benchmark sets. We leveraged R-Droid [7], a static IFC tool that does not interfere with our IAC/ICC model, to generate the intra-app flows[6], and compared our analysis results with the provided ground truth. We also verified the analysis results manually. As expected IIFA resolved our own tests with key and type matching correctly (i.e. without false positives).

Our design of IIFA as a pre-analysis does not negatively impact precision and soundness (even when factoring out precision gains due to R-Droid) but enables precise matching not only of intent actions but also of the key and/or type of intent extra data without additional constraint solving to exclude infeasible flows.

Clearly these benchmarks, three of which are gathered from related work, do not involve typically ignored (as hard to analyze) language features like reflection, or native code. However, they do cover a range of difficult features with respect to intent analysis, like dynamically composed strings, arrays with both sensitive and insensitive data, etc. The intention of this research question was to assert that the scalability gains (see RQ2) do not negatively impact other essential properties of our analysis, and to demonstrate the impact of precise matching of types and keys for transmitted intent data. Even though small, this microbenchmark evaluation demonstrates that our pre-analysis is an effective approach for ICC/IAC analysis.

IIFA + FlowDroid, AmanDroid & IccTA As of now, IIFA's resolution requires intra-component information flows (program slices) in Smali format. To validate the effectiveness of IIFA's pre-analysis approach with other tools that support intent-based information flow resolution (without requiring Smali slices), we created mutated APKs where the API call to receive data via intents is replaced by the actual data transmitted by the intent sender based on resolving ICC/IAC paths using IIFA's database. We attempted to analyze these APKs with FlowDroid (with added ICC analysis), AmanDroid and IccTA. As these tools create their own data flow paths for intent resolution, IIFA cannot improve the false positive scenarios. However, IIFA significantly improves the false negatives to true positives. It increases the

[6]Note that any other tool that resolves intra-component flows (in particular those of Table 4.3 except for DIALDroid) would also have been a possible base analysis, but may have interfered with our ICC/IAC model (see section 4.5.4).

Table 4.4: IAC analysis support vs Android API

Tool	IAC support	API 19	>API 19
AmanDroid	✓	✓	–
DidFail	✓	– ↯	– ↯
AppScan	×	–	–
DroidSafe	✓	✓	– ↯
DIALDroid	✓	✓	↯
FlowDroid	×	–	–
IccTA	✓	✓	– ↯

✓ supported, – fails, ↯ crashes

✓ supported, × not supported
✓ Not supported by default, additional configuration required

recall rate of FlowDroid from 80 to 91% and of AmanDroid from 60 to 85%. In few cases (e.g., bindService2) FlowDroid and AmanDroid still report a false negative because their analysis computes an incomplete call graph of the callback method *onServiceConnected*. Unfortunately IccTA is unable to analyze our modified APKs.[7] These experiments demonstrate the high quality of our intent model.

4.5.2 Evaluating the scalability of IIFA

Evaluation on real-world apps

We applied IIFA to real-world apps to assess whether our analysis scales to a realistic corpus of large real-world apps. To that end we downloaded the 90 most popular apps from the Google playstore, which arguably contain some of the most challenging apps for program analysis, e.g. due to their size. IIFA successfully analyzes each of these apps. Table 4.4 (extended from [64, 66]) lists the related works along with their IAC analysis capabilities in general and for API levels of Android. It is important to observe that none of the tools (that support IAC analysis) is able to analyze an app using API version greater than 19 (i.e. Android KitKat). As this Android version was released in 2013, it is almost impossible to find APKs amenable for analysis by these tools, rendering them obsolete for IAC analysis of realistic Android apps. Even the authors of several related tools admit that 'DroidSafe, IccTA and ApkCombiner all crash while analyzing apps built for an API above 19, which is supported by the majority (82.3%) of Android devices. A common cause is a tool-dependency on [...] Apktool. Old versions of it fail to decompile newer Android

[7] IccTA throws an *invalid magic value* error when analyzing any APK modified by our APKTool. We assume version incompatibility between the IccTA's *dexlib (2-2.1.0)* and APKtool's *dexlib (2-2.2.3)*.

apps. The same happens to [...] ApkCombiner. [Therefore] Amandroid, DroidSafe, FlowDroid and IccTA lose their ability to analyze inter-app scenarios.' [64]. While these issues might be technically solvable with some engineering effort, we will argue in the sequel, that only pre-analysis is scalable enough to analyze the IAC information flows for all apps installed on a given device.

As 60% of all intents are implicit, analyzing IAC flows becomes paramount to detect hidden or accidental leaks of sensitive data contrary to user expectations. ApkCombiner was proposed to merge APKs in order to extend standard ICC analysis mechanisms (where all potential communicating components are in one APK) to IAC. As mentioned in the previous paragraph, APKCombiner fails for practically all relevant APKs. But even if merging was possible (some related work merges directly in their tools) analyzing the resulting APK faces combinatorial explosion of potential communication paths and would require additional constraint solving technologies to prune unrealizible inter-component data flow paths due to mismatching communication partners (e.g. intent action) or keys/types of the data exchanged via intend extra data.

In contrast IIFA propagates a divide-and-conquer approach where ICC/IAC communication partners are determined and constraints solved in a pre-analysis based on summary information extracted from each app in isolation. The base IFC analyses then only need to provide intra-component information flows (program slices), which is what most of these tools have originally been designed and leverage intricate optimizations for. To demonstrate the advantage of the pre-analysis approach we created a potential IAC flow from a widely used real-world app *Katwarn* to a synthetic app (written using target API level 19 to enable analysis by these tools). One of *Katwarn*'s activities (*GuardianAngelService*) shares the last known location via an implicit intent. An app that declares the corresponding intent filter (*kwrn:ga:location:update*) may receive this intent. Our synthetic app declares the *kwrn:ga:location:update* intent filter to illicitly intercept this intent. Listing 4.7 shows a simplified version of the receiving activity in this test app. Analyzing this IAC without IIFA failed as APKCombiner was incapable of merging these two apps, and tools that create their own paths, e.g. DIALDroid, crashed. However, when analyzing the test app in isolation, all of these tools produce the output slice for the test app, which resembles the smali code in Listing 4.8. Using IIFA's analysis DB and including that slice into the reporting module (cf. Section 4.4.2), detecting

```
1
2 // (ReceivingActivity) -- Capable of receiving
      "kwrn:ga:location:update"
3 Bundle bundle = getIntent().getExtras();
4 Location lastKnownLocation = (Location)bundle.get("location");
5 Log.d("Location", location.toString());
```

Listing 4.7: ReceivingActivity

```
1 invoke-virtual {p0},
      Lcom/testapplication/ReceivingActivity;->getIntent()
      Landroid/content/Intent;
2 move-result-object v0
3 invoke-virtual {v0},
      Landroid/content/Intent;->getExtras()Landroid/os/Bundle;
4 move-result-object v0
5 .local v0, "b":Landroid/os/Bundle;
6 const-string v1, "location"
7 invoke-virtual {v0, v1},
      Landroid/os/Bundle;->get(Ljava/lang/String;)Ljava/lang/Object;
8 move-result-object v1
9 check-cast v1, Landroid/location/Location;
10 .local v1, "location":Landroid/location/Location;
11 const-string v2, "Location"
12 invoke-virtual {v1},
      Landroid/location/Location;->toString()Ljava/lang/String;
13 move-result-object v3
14 invoke-static {v2, v3},
      Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
```

Listing 4.8: ReceivingActivityIRinSmali

the mentioned IAC flow is thus possible, rendering IIFA a practical, effective, and scalable addition to the state-of-the-art tools.

IIFA's scalability vs Merging-based analysis Every tool except DidFail and DIALDroid requires merging of APKs to extend their ICC capabilities to IAC. To analyze inter-app communication, *ApkCombiner* [52] was proposed [52] in order to merge two or more apps into one. However, *ApkCombiner* supports only Android app versions published on or before 2014, and crashes with the recent apps, thus being practically unusable. But even if the merging process itself was not problematic, it would aggravate the scalability issues reported for related work [53], and confirmed in a independent recent comparative study [65]. Practically attempting to analyze a huge APK with all of the top 90 apps from the *Google Play Store* would lead to a com-

binatorial explosion of communication paths between potentially communicating components to be analyzed, precluding any precise static analysis. In particular as we found in our study with the top 90 apps from the *Google Play Store* that 60% of intents are implicit and thus the receiver may not be unique.

Alternatively one would have to eagerly merge all combinations of (at least) two complete apps. This is at least 8,100 combinations for our 90 apps already, most of which are not communicating. However, this approach assumes that there is no communication involving more apps than the maximum tuple size. Unfortunately there is no guarantee for this assumption unless analyzing all tuples of larger size (which does not scale any more).

In contrast, IIFA analyzes the communication compositionally based on small summaries in a database and combines only the transmitted ICC/IAC data with the intra-app flows of respective intent receivers. The average per app execution time of IIFA over 90 apps is 87.91 seconds. On average, the analysis phase took 46.81 seconds (maximum 52.20, minimum 32.40 seconds) and the reporting phase 41.10 seconds (maximum 48.60, minimum 35.10 seconds). Considering that (intra-app) static IFC analyses usually require a large amount of time to analyze real world apps, IIFA's additional cost is quite feasible for a realistic usage scenario. Let us take 30 minutes as the execution time [64] require by every tool to analyze a single real world app. IIFA adds approximately 1.5 minute per app. We refer to this sum of t_{IFC} and the time taken by IIFA as t_{sum}. Thus, the total time taken by IIFA for 90 apps is $90 \times t_{sum}$ ($\approx$ 2 days). Whereas, merging all pairs would take $90 \times 90 \times t_{IFC}$ ($\approx$ half a year) to analyze ICC between any two apps, let alone tuples of greater size. Clearly, the total time is dominated by the combinatorial explosion of merging apps. Therefore, IIFA would take at most 1.2% of the time taken by idea to analyze all merged pairs.

IIFA's modular analysis avoids combinatorial explosion of the potential flow paths in a single merged APK (containing all potential communication partners), or of analyzing all tuples of a given size. IIFA's resolution of intent actions, keys and types for intent extra data additionally reduces the analysis complexity without requiring supplementary constraint solving technology during ICC information flow analysis. Therefore we were able to analyze the ICC/IAC flows of the 90 most downloaded real-world apps in approx. 2.2h (ignoring the time to compute intra-component slices).

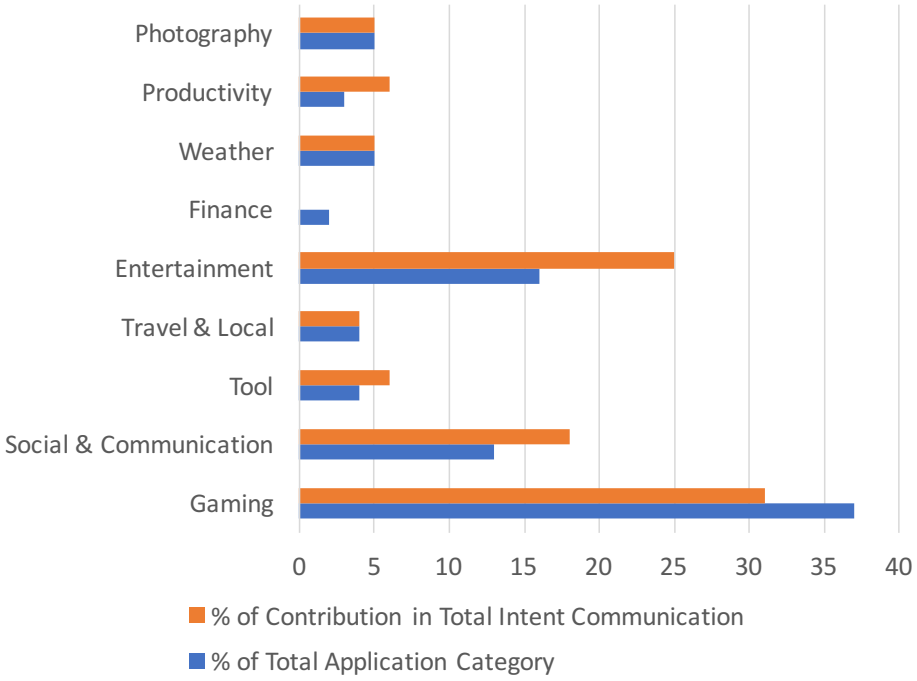


Figure 4.3: Intent Usage

4.5.3 Communication patterns in real-world apps

In total we identified 10,669 calls to start an intent (either as an Activity, Broadcast or Service), 76% of these leverage *startActivity/startActivityForResult*. Further, the data in IntentDB show that (statically) 60% of the intents are implicit. The Android documentation defines 42 different types of *getXXXExtra* methods. IIFA determines that 54.6% of these methods retrieve String values, either via *Bundle.getString(String)* (38.8%) or *String.getStringExtra(String)* (15.7%). For 2% of the sent intents IIFA was unable resolve the target component or intent action and conservatively approximated it to either multiple actions (0.6%) or even a dummy action (1.4%), the latter of which requires manual inspection to resolve the potential strings (e.g. due to dynamic input from a file).

Figure 4.3 provides a categorization of the 90 apps along with their intent usage. 37% of the apps are *Games*, which contribute the most to intent communications (31%). Interestingly, we find that 6% of the total intent communication is triggered by a single a *Communication* app, *whatsApp*. It contributes to 35% of the intent communication in the *Communication* category.

IIFA's database contains senders with 380 distinct Intent actions, 100 out of them

with corresponding receivers in our test set (excluding system apps and OS). Figure 4.4 displays the 20 most used intent actions and their numbers of senders and receivers, the remaining intent actions had only one or tow senders and receivers. Note that the numbers of *actual* intent receivers is lower as the type and key matching of intent extra data must also match. *android.intent.action.VIEW* is most widely used (73 senders and 52 receivers) followed by *android.intent.action.SEND* (50 senders and 15 receivers). *android.intent.action.VIEW* is used to display the data to the user, e.g., an activity that wants to open a webpage would create an intent with this target action and a browser activity would register this action as the supported intent filter in its manifest file. Similarly, *android.intent.action.SEND* is used to send some data from one activity to another, e.g., to send an email an activity would create an intent with this target action and a receiving activity would define this intent filter in its manifest file. *android.intent.action.DIAL* is an Intent action to invoke the OS phone dialer. As *Viber*, a *voip* app, also registers to receives it, users will be asked to select how to make a phone call. Other apps not providing *voip* ought not receive it. IIFA determines potential rogue apps via a simple database lookup. To the best of our knowledge, we are the first to predict and analyze these communication patterns.

IIFA detects 62 ICC-based information flows from sensitive sources among the 90 apps. Our results shows that even widely used apps share sensitive information via implicit intents, which may lead to *intent interception* and *intent hijacking* attacks [58]. We manually validated these claims in the following apps: *Katwarn* provides hazard and disaster warnings and has been downloaded over one million times. IIFA finds that one of its activities (*GuardianAngelService*) shares the last known location via an implicit intent. An app that registers the respective intent filter can try to intercept this intent (*intent interception*) to obtain the device's location without the respective permission. Similarly, the shopping app *ebay* (via activity *EventItemsFragment*) and the location & travel app *Google Earth* share internal device resources via an implicit intent, which are thus also prone to intent interception. IIFA also determines that *ebay* shares sensitive *tags* via both implicit (in *CheckoutActivity* activity) and explicit intents (in *PaypalCreditPromotionsActivity* activity). We also identified some cases of potential data integrity violations: For example, a *tag* can contain a (non-sensitive) *url*, e.g., BankA.com, to be opened by an intent receiver. If this *tag* is shared via an implicit intent, a malicious app that registers the respective intent filter may manipulate this *tag* to open another url, e.g., BankB.com (*intent hijacking* as part of

a phishing attack). While identifying potential security issues was not the main intent of this work, we notified the corresponding developers and suggested the required fix to make their app more security compliant.

IIFA analyzes the IAC patterns and may detect rogue apps registering for Intent actions they are not supposed to handle. We detect a number of problematic information flows that have not been reported previously and may be abused by malevolent apps.

4.5.4 Evaluation Summary and Discussion

We empirically evaluated IIFA in two steps. IIFA does not suffer from scalability issues (RQ2) as it analyzes each app once (with potentially one additional intra-app IFC analysis by an external tool), and its precision and recall are not negatively impacted by this design.

Due to the nature of our analysis we rely on program slices generated from a baseline (intra-component) IFC analyzer. Therefore our combined analysis as presented in the evaluation section inherits all advantages and disadvantages as well as potential implementation bugs of the underlying analysis. In order to rule out any potential interference of our analysis with the baseline analysis we chose R-Droid as our baseline analysis as it is relatively precise but does not attempt to resolve intent communication on its own. At the same time as hardly any analysis tool considers the grand obstacles for static program analysis like native code and reflection, our combined analysis also lacks these features, which we consider largely orthogonal to this line of research. Further, we concentrate on intent-based communication in this work and ignore other, more atypical forms of inter-component communication such as static (global) variables or content providers. Some baseline analyses support those, in which case a combination of IIFA with such a tool would also do.

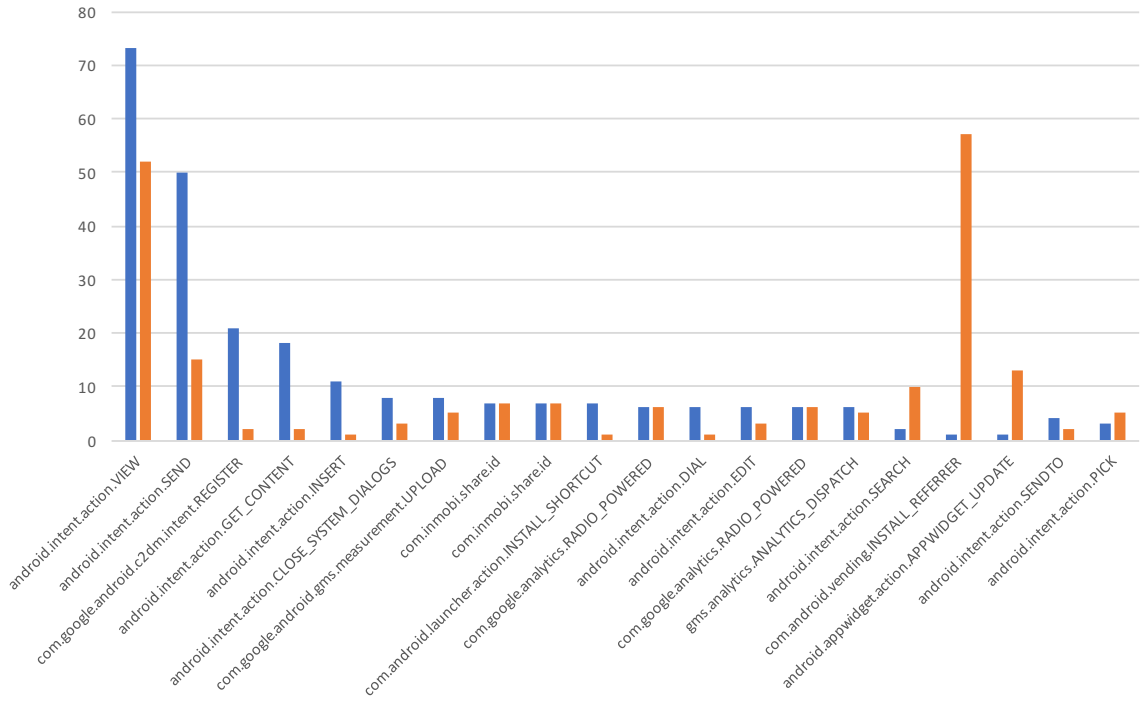


Figure 4.4: Intent Actions (x-axis) per sender (blue) and receiver (orange, y-axis)

Chapter 5

PIAnalyzer: A precise approach for PendingIntent vulnerability analysis

5.1 Overview

Android phones are used for a plenitude of highly security critical tasks, and a lot of sensitive information—including session tokens of online services—are saved on these devices. As the official Google Play Store and other alternative Android app marketplaces are not strongly regulated, the main defense against malware that aims to steal sensitive information is the Android sandbox and permission system. In our study we discover that the Android permission system can be circumvented in many cases in the form of denial-of-service, identity theft, and privilege escalation attacks. By exploiting vulnerable but benign applications that are insecurely using PendingIntents, a malicious application *without any permissions* can perform many critical operations, such as sending text messages (SMS) to a premium number. PendingIntents are a widespread Android callback mechanism and reference token. While the concept of PendingIntents is flexible and powerful, insecure usage can lead to severe vulnerabilities. Yu et al. [83] report a PendingIntent vulnerability in Android's official Settings app, which made a privilege escalation attack up to SYSTEM privileges possible for every installed application. Thus, given the severe security implications, the official Android documentation on PendingIntents [27]

now warns against insecure usage. However, to the best of our knowledge, to-date no analysis tool detects the described PendingIntent vulnerabilities. Thus, an automated analysis tool is envisioned that scales to a large number of applications.

5.2 Our Contributions

In this work we propose a novel approach to detect PendingIntent related vulnerabilities in Android applications. We implemented our approach in a tool called PIAnalyzer. In multiple analysis steps, PIAnalyzer computes the relevant information of the potentially vulnerable code based a program slicing [85]. PIAnalyzer is fully automated and does not require the source code of the application under inspection. PIAnalyzer assists human analysts by computing and presenting vulnerability details in easily understandable log files. We evaluated PIAnalyzer on 1000 randomly selected applications from the Google Play Store. We discover 435 applications that wrap at least one implicit base intent with a PendingIntent object, out of which 1358 insecure usages of PendingIntents arise. These include 70 PendingIntent vulnerabilities leading up to the execution of critical operations from unprivileged applications. We manually investigate multiple findings and inspect reports on examples known to be vulnerable. Our investigation show that PIAnalyzer is highly precise and sound. Technically, we provide the following contributions:

- *PendingIntent analysis.* We propose a novel method based on program slicing for the detection of PendingIntent related vulnerabilities.
- *Implementation.* We implemented a program slicer for SMALI intermediate code and the proposed PendingIntent analysis in a tool called *PIAnalyzer*.
- *Evaluation of PIAnalyzer.* We empirically evaluate PIAnalyzer on a set of 1000 randomly selected applications from the Google Play Store and find 1358 insecure usages of PendingIntents. These include 70 severe vulnerabilities. We find critical vulnerabilities in widely used libraries such as, *TapJoy* and *Google Cloud Messaging*. PIAnalyzer is efficient and only takes 13 seconds per application on average.
- *Validation of PIAnalyzer.* We manually validated multiple reports of PIAnalyzer. Our validation confirms PIAnalyzer's high precision and recall.

```
1 protected void onCreate(Bundle savedInstanceState) {
2    super.onCreate(savedInstanceState);
3    setContentView(R.layout.activity_main_vuln);
4    Intent baseIntent = new Intent();
5    PendingIntent pendingIntent = PendingIntent.getActivity(this,
      1, baseIntent, PendingIntent.FLAG_UPDATE_CURRENT);
6    Intent implicitWrappingIntent = new Intent(Intent.ACTION_SEND);
7    implicitWrappingIntent.putExtra("vulnPI", pendingIntent);
8    sendBroadcast(implicitWrappingIntent);
9 }
```

Listing 5.1: App A - Vulnerable Activity

```
1 public void onReceive(Context context, Intent intent) {
2    Bundle extras = intent.getExtras();
3    PendingIntent pendingIntent = (PendingIntent)
      extras.get("vulnPI");
4    Intent vunlnIntent = new Intent(Intent.ACTION_CALL,
      Uri.parse("tel:" + "0900123456789"));
5    try {
6       pendingIntent.send(context, 2, vunlnIntent, null, null);
7    } catch (PendingIntent.CanceledException e) {
      e.printStackTrace(); }
8 }
```

Listing 5.2: App B - Malicious Activity

5.3 A Motivating Example

5.3.1 A Potential Vulnerable Example

We demonstrate PendingIntent-related vulnerabilities and exploitation via a simplified example (Listings 5.1 and 5.2). In this example, the vulnerable application has the permission to perform phone calls, while the malicious application does not. In listing 5.1, the vulnerable application creates an empty base intent (line 4), wraps it into a PendingIntent (line 5), and sends it as an extra of the broadcast intent *implicitWrappingIntent* (line 6, 7, 8). Any application that defines a corresponding intent filter in their manifest file can receive *implicitWrappingIntent*. In listing 5.2, a malicious application which is capable of receiving *implicitWrappingIntent*, extracts the PendingIntent (line 3) and creates a new intent (with the motivation to manipulate the base intent) (line 4) such that it triggers a phone call to some arbitrary number, e.g., to a premium number. On line 6, an invocation of the *send* method of this PendingIntent object causes the execution of the base intent but with all empty

properties updated to the values specified in *vulnIntent*, which results in calling the premium number.

While this example is simplified for better understanding, PendingIntent vulnerabilities can occur in various forms and lead to different types of severe security implications. In our study we find that at least 435 out of 1000 applications wrap at least one implicit base intent into a PendingIntent object. The vulnerable application can accidentally send the PendingIntent object in numerous ways. Instead of broadcasting, it can also be sent out via an implicit wrapping intent. If more than one application has a matching intent filter, the user will be asked to choose a destination. This case can be abused by an intent phishing application. In the majority of the cases the PendingIntent is not sent out by wrapping it into another intent, but by passing it to system components such as the AlarmManager or the NotificationManager. These components will eventually call the *send* method of the PendingIntent object, which triggers the base intent. A malicious app can register a component to retrieve the base intent to perform a denial of service attack, as these intents are then not passed to the intended component.

This situation becomes even more critical when the described PendingIntent vulnerability occurs in system components. Tao et al. [83] found this type of vulnerability in the Android Settings application. In the following subsection we elaborate on the details of this vulnerability.

5.3.2 A Real-world PendingIntent Vulnerability

For all subversions of Android 4, the *Settings* application triggered a PendingIntent with an empty base intent [83]. In February 2018, 17.4% of all Android devices still run such an Android version [34], rendering them vulnerable to a privilege escalation attack up to system privileges. This vulnerability occurred due to unawareness of the PendingIntent security implications. The fix in Android version 5.0 makes the base intent explicit.

Listing 5.3 shows the code snippet of the corresponding vulnerable method *addAccount*. In this method a PendingIntent object, *mPendingIntent*, is created (line 3) with an empty base intent. Whenever an application requests to add an account of the requested (custom) type, the *addAccount* method gets invoked and the vulnerable PendingIntent (*mPendingIntent*) is returned to this application if it registers

```
1 private void addAccount(String accountType) {
2    Bundle addAccountOptions=new Bundle();
3    mPendingIntent=PendingIntent.getBroadcast(this, 0, new
      Intent(), 0);
4    addAccountOptions.putParcelable(KEY_CALLER_IDENTITY,
      mPendingIntent);
5    addAccountOptions.putBoolean(EXTRA_HAS_MULTIPLE_USERS,
      Utils.hasMultipleUsers(this));
6    AccountManager.get(this).addAccount( accountType, null,
7       /* authTokenType */ null, /* requiredFeatures */
           addAccountOptions,
8       null, mCallback, null /* handler */);
9    mAddAccountCalled = true;
10 }
```

Listing 5.3: Android Settings: AddAccountSettings.java

```
1    Intent intent = new Intent();
2    intent.setComponent(new ComponentName("com.android.settings",
      "com.android.settings.accounts.AddAccountSettings"));
3    intent.setAction(Intent.ACTION_RUN);
4    intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
5    String authTypes[] = {AccountGeneral.ACCOUNT_TYPE};
6    intent.putExtra("account_types", authTypes);
7    startActivity(intent);
```

Listing 5.4: Malicious Application A: Activity 1

to receive `android.accounts.AccountAuthenticator` intents (see Listing 5.5). As this application executes *mPendingIntent* in the context of the *Settings* application (with *SYSTEM* level permissions), it can maliciously overwrite the (empty) action and extra data in the base intent.

Listing 5.4 and 5.5 describe the code snippets of a malicious application *A*, targeting the vulnerability of the *Settings* application. In listing 5.4, *A* initiates an intent to add an account type (line 7). Upon reception of this intent, the *Settings* application invokes the *addAccount* method (cf. Listing 5.3) and sends *mPendingIntent* out. As *A* has registered as *AccountAuthenticator*, it receives this PendingIntent (line 2 of Listing 5.5). On line 3, it creates an intent *vunlnIntent* to perform a Factory Reset[1]. Later it triggers the PendingIntent with *vunlnIntent* as the updated base intent (line 5). As *A* executes the PendingIntent in the same context as the *Settings* application (with *SYSTEM* level permissions), a Factory Reset is performed.

As previously described, the key cause of this type of vulnerability is the usage of

[1] A factory reset resets the device to its factory setting, i.e., deletes all data.

```
1 public Bundle addAccount(AccountAuthenticatorResponse response,
    String accountType, String authTokenType, String[]
    requiredFeatures, Bundle options) throws NetworkErrorException
    {
2 PendingIntent pi =
    (PendingIntent)options.getParcelable("pendingIntent");
3 Intent vunlnIntent = new
    Intent("android.intent.action.MASTER_CLEAR");
4 try {
5   pi.send(mContext, 0, newIntent2, null, null);
6 } catch (CanceledException e) { e.printStackTrace(); }
```

Listing 5.5: Malicious Application A: Activity 2

implicit base intents for PendingIntents. Therefore, in this work we provide a novel analysis mechanism which detects implicit base intents in PendingIntents, analyzes their usage and gives a security warning in case of an actual vulnerability.

5.4 Methodology

5.4.1 SMALI and SMALI Slicing

PIAnalyzer analyzes the SMALI intermediate representation (IR) of the Dex bytecode extracted from an APK. SMALI is an intermediate representation of Dalvik bytecode that improves readability and analyzability. As background information, Listing A (in the appendix) shows a simplified example of the creation of an Intent object in SMALI code. Similar to Dalvik bytecode, SMALI is register based. As known from assembly languages, registers are universally used for holding values. For example, on line 15 the register *v3* is used to store a String variable, while on line 18 an Intent object is saved in the register *v0*. Please consider the comments in the listing for a more detailed explanation of the code.

PIAnalyzer transforms the bytecode of an APK to its SMALI IR using APKTool [1]. The core of the analysis of PIAnalyzer is performed through program slicing [85] the SMALI representation. Conceptually, a slice is a list of statements that influence a statement (backward slice), or get influenced by a statement (forward slice). For this purpose we design a SMALI slicer. Our SMALI slicer can create both forward and backward slices that are required for the analysis of PIAnalyzer. As registers are SMALI's universal storage mechanism for holding any kind of values, our SMALI

Figure 5.1: The workflow of PIAnalyzer

slicer is register based. The SMALI slicer is initialized with an arbitrary start position in the code as well as with a set of relevant registers. After completion it returns a set of influencing statements. For example, in Listing A the backward slice of the registers *v0* and *v3*, starting from line 21 will return the statements on lines 15, 18 and 21 as backward slice. We would like to stress that the PendingIntent analysis described in the following is just one usage of our developed slicer. In fact, our slicer is universal and can be used for various program analysis purposes. The software architecture of PIAnalyzer is designed in a modular way that facilitates the extension by further analysis approaches. Similar to the analysis of PendingIntents, these approaches can easily make use of our generic SMALI slicer.

5.4.2 PendingIntent Analysis

PIAnalyzer is designed for the efficient analysis of a large number of APKs and therefore accepts as input an arbitrarily large set of APKs. Figure 5.1 depicts the workflow of PIAnalyzer per APK. The analysis of PIAnalyzer consists of the following steps.

PendingIntent extraction. PIAnalyzer decompiles the DEX bytecode of a given APK to the SMALI IR using APKTool [1]. It then parses the content of each SMALI file together with the application's manifest. PendingIntents can only be created by four methods: *getActivity()*, *getActivities()*, *getBroadcast()* and *getService()* [27]. PIAnalyzer searches in the parsed SMALI files for calls to these methods, leading to a complete list of all PendingIntent creations in the application.

Base Intent analysis. In the next step PIAnalyzer extracts the base Intent object used for creating the PendingIntent. It builds the backward slice from the PendingIntent creation site to the creation site(s) of the base Intent leveraging our universal SMALI slicer. Based on this backward slice, PIAnalyzer determines whether the base Intent is potentially implicit, meaning no target component was definitely set. For determining whether an Intent may be implicit, PIAnalyzer first confirms that

an implicit constructor, i.e., a constructor without a specified target component was used to create the base Intent. It then examines whether an explicit transformation method was invoked on the base Intent object. Explicit transformation methods set the target component of an Intent object after it has been constructed, transforming an implicit Intent into an explicit one. To the best of our knowledge only five explicit transformation functions exist at the time of this writing: *setClass()*, *setClassName()*, *setComponent() setPackage()* and *setSelector()*. If an Intent has been created by an implicit constructor and no explicit transformation has definitely been invoked on the Intent, it is considered implicit. In the following steps, PIAnalyzer only considers occurrences of PendingIntents with implicit base Intents, as only these can lead to the described security issues (see discussion in section 2.3.2).

PendingIntent analysis. The severity of the vulnerability depends on the usage of the PendingIntent. Concretely, it depends on the sink functions to which the PendingIntent object is passed. A PendingIntent can either be sent to a trusted system component, e.g. Alarm manager, or wrapped into another Intent. PIAnalyzer therefore computes the forward slice from the creation of the PendingIntent object to either of the mentioned APIs, using our universal SMALI slicer.

WrappingIntent analysis. The most dangerous class of attacks can occur if the PendingIntent object itself is intercepted by a malicious application. This can happen if the PendingIntent is wrapped in another intent (referred to in the sequel as *wrapping Intent*) as Intent extra data. If the wrapping Intent is implicit it can be received by a malicious application to extract its wrapped PendingIntent and manipulate the base Intent. To detect this particularly dangerous class of vulnerabilities, PIAnalyzer examines all wrapping Intents whether they are implicit, as only in this case they can be received by a malicious application. To that end, PIAnalyzer creates the backward slice for all wrapping Intents using our universal SMALI slicer. From the resulting slice it determines whether the wrapping Intent is implicit (in analogy to the base Intent analysis phase), in which case it reports a vulnerability.

Call Graph generation. PIAnalyzer is designed to facilitate the analysis of human security experts. PIAnalyzer assists human investigation of a reported vulnerability via a generated the call graph, which leads to the the method in which it has been detected. Thus human experts may determine the events that lead to the execution of the vulnerable code spot. The generated call graph can track control flow between the main application and its used libraries. Additionally, it handles recur-

sive functions.

Reporting. In the last phase, PIAnalyzer logs the results of the analysis. PIAnalyzer creates two types of log files: For each detected vulnerability, it creates a vulnerability log file that reports details of that vulnerability. Additionally, it creates a summary log file that summarizes the findings in the whole APK batch and gives general statistics:

- *Vulnerability Log File* This file contains the slice from the creation of the base intent, over the creation of the PendingIntent object, to the final sink function. Additionally, each vulnerability log file contains the slice from the base Intent to the PendingIntent, as well as the PendingIntent forward slice. Finally, the call graph to the method containing the vulnerability is logged.

- *Summary Log File* For each batch of APKs one summary log file is created. Apart from some hardware specifications, this summary log file contains the total number of warnings and vulnerabilities, as well as their some statistics over the batch of APKs.

5.4.3 Vulnerability severity levels

PIAnalyzer distinguishes the following levels of severity (in increasing order):

- *Secure* PendingIntents with explicit base Intents are considered secure as a known and apparently trusted component is invoked. We respect this trust relation and create no report for these cases.

- *Warning* If a PendingIntent with an implicit base Intent is created, but this PendingIntent is only passed to System managers that are supposed to be benign, it is considered a *Warning*. As a System manager will not redefine the base Intent of the PendingIntent, the only possible attack scenario in this case is a denial of service attack if a malicious application catches the implicit base Intent after the System manager has triggered the *send()* method of the PendingIntent.

- *Vulnerability* PIAnalyzer reports a *Vulnerability* if a PendingIntent has been created with an implicit base Intent and the PendingIntent has been wrapped

Table 5.1: Distribution of vulnerabilities and warnings

# vuln.	0	1	2	4
# apps	938	56	5	1

# warn.	0	1	2	3	4	5	6	7	8	9	10	13
# apps	565	104	101	96	61	33	16	16	2	2	3	1

in an another implicit WrappingIntent. In this scenario a malicious application can receive the PendingIntent, and redefine its base Intent resulting in a privilege escalation attack.

5.5 Evaluation

We applied PIAnalyzer to 1000 randomly selected applications from the Google Play Store. All experiments were performed on a MacBook Pro with MacOS High Sierra 10.13.3 installed, a 2,9 GHz Intel Core i7 processor and 16 GB DDR3 RAM.

PIAnalyzer reports 70 PendingIntent vulnerabilities and 1288 PendingIntent warnings[2]. We statistically analyzed the distribution of vulnerabilities and warnings among the inspected applications. Table 1 depicts the distribution ratios. In the vast majority of the cases a vulnerability does not occur more than once per application. However, the situation is different for warnings. Our findings show that it is likely for an application to include more than one warning. PendingIntents are thus more likely to be delivered to system components (e.g., AlarmManager).

Additionally, we analyzed the proportion of vulnerabilities and warnings that were contained in third party libraries. Remarkably, we find that 80% of the reported vulnerabilities and 98% of the reported warnings occur in third party libraries. Third party libraries thereby act as a multiplier for vulnerabilities, as they are used by a large number of applications. We therefore would like to stress the importance of PIAnalyzer for library developers. Table 5.2 provides a list of these libraries along with their contribution to the number of vulnerable apps. Libraries are included as the dependencies in the *build.gradle* file [3]. As this file is not compiled into the APKs, we could not find the exact version of the library. A tedious way to find the exact version of the library is to match the app's intermediate code with

[2]For explanations of the severity levels please refer to section 5.4.3
[3]https://developer.android.com/studio/build/index.html

Table 5.2: Libraries contribution to number of vulnerabilities

Library	Description	app vuln.	Year
Google Messaging Library	Cloud Messaging	39	2017
Cloud to Device Messaging	Cloud Messaging	8	2016
TapJoy	Marketing and Automation	3	2016
MixPlane	Push Notification & In App Messaging	4	2016
LeanPlum	Messaging, Variable, Analytics & Testing	2	2017

the intermediate code of the each version of library. We find that these versions of libraries are still in use in the recent versions of applications. Thus, instead of providing the exact version of libraries we provide their year of appearance in an application (in 1000 applications from our experiment).

As mentioned, an attacker can escalate a PendingIntent vulnerability into a privilege escalation attack and leverage the permissions of the vulnerable applications. We therefore analyzed the permissions of the applications for which PIAnalyzer reported vulnerabilities. We find that 279 dangerous permissions [35] and 273 normal permissions are used by these vulnerable applications. As dangerous permissions are required for performing critical operations on the device, an attacker may act maliciously in many of these instances, e.g., call a premium number.

Table 5.3 provides a list of ten vulnerable applications (randomly selected) along with their category and used dangerous permission groups[4]. The permission groups contain permissions organized into a device's capabilities or features, e.g., *PHONE* group includes the *CALL_PHONE* permission. 35% of the vulnerable applications belong to the *Business, Entertainment, or Education* category.

In our experiment with 1000 real-world applications, the average execution time of PIAnalyzer is close to 13 seconds with a minimum of 10 seconds and the maximum time of 21 seconds. This time performance strongly demonstrates the efficiency of PIAnalyzer and proves that it can easily be applied to a large number of real-world applications.

To evaluate the precision and the soundness of PIAnalyzer, we manually inspected the reported results of ten applications. Manual inspection is time consuming as it requires analysis of many SMALI code files. Out of ten applications, we find that nine times PIAnalyzer reports correct vulnerabilities/warnings, indicating a

[4]Due to the lack of space, we do not provide the full list of 1000 apps nor exact permissions here. We will publish the full list after the publication of this paper together with the source code of our analysis.

Table 5.3: Vulnerable applications with dangerous permissions

App Name	App Category	Dangerous Permission Group
SandWipPlus	Communication	Contacts, Phone, Sms, Storage
Reason	News & Magazines	Contacts, Location, Phone, Storage
Santa Dance Man	News & Magazines	Phone, Storage
SmartInput Keyboard	Personalization	Phone
drift15house	Entertainment	Calendar, Contacts, Location, Phone, Storage
Fishermens	Entertainment	Contacts, Camera, Location, Microphone, Phone, Storage
Derek Carroll	Photography	Camera, Location, Microphone, Phone, Photography, Sms, Storage
ElleClub	Business	Camera, Contacts, Location, Microphone, Phone, Sms, Storage
Chat Locator	Productivity	Location, Storage
Deptford Mall	Lifestyle	Calendar, Contacts, Location, Microphone, Phone, Sms, Storage

high precision. In one case, the base intent was manipulated dynamically and thus PIAnalyzer conservatively overapproximated it as implicit intent.

In addition, we applied PIAnalyzer to the vulnerability in the *Settings* app (described in the section 5.3.2) that led to privilege escalation to SYSTEM privileges. PIAnalyzer correctly reports the vulnerability and so PIAnalyzer could have prevented the discussed vulnerability. Finally, we applied PIAnalyzer to multiple self-written demo examples that included PendingIntent vulnerabilities. PIAnalyzer correctly reports each of them, indicating high recall.

5.5.1 Case Study: Vulnerability in the Google Cloud Messaging (GCM) Library

PIAnalyzer finds a vulnerability in an outdated version of the Google Cloud Messaging (GCM) Library, which is part of the Google Messaging Library and still in use by many applications, e.g., Table Tennis 3D [56] or the Android Device Manager. Among 1000 analyzed applications, we find that 37 out of 39 (cf. Table 5.2) applications still use this version of GCM. The vulnerability exists in the file *GoogleCloudMessaging.java* of the GCM Library. Listing 5.6 shows the code snippet of the vulnerable method *send*. On line 4, an implicit intent named *localIntent* is created.

```
1 public void send(String paramString1, String paramString2,
2         long paramLong, Bundle paramBundle) {
3    // ...
4   Intent localIntent = new
      Intent("com.google.android.gcm.intent.SEND");
5   localIntent.putExtras(paramBundle);
6   c(localIntent);
7   localIntent.putExtra("google.to", paramString1);
8   localIntent.putExtra("google.message_id", paramString2);
9   localIntent.putExtra("google.ttl", Long.toString(paramLong));
10  this.eh.sendOrderedBroadcast(localIntent, null);
11 }
```

Listing 5.6: Vulnerable Method

```
1 void c(Intent paramIntent) {
2    try {
3       if (this.xg == null)
4          this.xg = PendingIntent.getBroadcast(this.eh, 0, new
      Intent(), 0);
5       paramIntent.putExtra("app", this.xg);
6    } finally {}
7 }
```

Listing 5.7: PendingIntent with an empty base intent

On line 6, *localIntent* is passed to a method *c*. Listing 5.7 shows the code snippet for the method *c*. In this method, a PendingIntent with an empty base intent is created (line 4). On line 5, this PendingIntent is stored as extra data to the input parameter *paramIntent*. Later in method *send* (listing 5.6), *localIntent* is broadcast to all registered receivers. Any Broadcast Receiver, declaring this intent filter (*com.google.android.gcm.intent.SEND*) in its manifest file, can receive this Intent and can easily extract the associated PendingIntent. In this case the permissions of the attacker application are escalated to the permissions of applications that use GCM. In our experiments, we are able to intercept *localIntent* and to extract the associated PendingIntent. As the base intent in the associated PendingIntent is blank, we set any arbitrary action/component and trigger it with the same identity as the vulnerable application. This enables us to perform arbitrary actions with the identity of the vulnerable application, e.g., sending a malicious message to a different component of the vulnerable application and making it believe it was sent from within the application (i.e. identity theft). In the worst case scenario, if this GCM version were used by a system application with system permissions (GCM is an official Google library), a malicious application could for example factory reset the device

(deleting all data). We tested several versions (4–7) of system APKs from Google without such inclusions found. However, due to lacking availability we could not check system APKs from other vendors.

5.6 Limitations

PIAnalyzer is a static analysis tool which shares common limitations with other static analysis approaches. As the program behavior can depend on dynamic input, every static analysis tool cannot be completely sound and precise. The slicing analysis of PIAnalyzer is affected by these limitations. In theory, it is possible to make an Intent implicit or explicit depending on external runtime input. This could happen either by making the constructor used for intent creation or the usage of explicit transformation methods depend on external input. When it is not clear at compile time whether an Intent is implicit or explicit, PIAnalyzer conservatively assumes that it is implicit. Analogously, the slices computed by PIAnalyzer are, by their nature, conservative approximations of the actual control flow at runtime. In theory, it is also possible to make use of Intents in Reflection or native code. PIAnalyzer neither supports reflection, nor native code. We would like to stress that while the above mentioned cases are possible in theory, they are rare in the real world and we could not observe a single instance of these cases during inspection of many applications. Some of the operations that require permissions cannot be performed directly via Intents to system interfaces, e.g., the retrieval of a precise location. Thus an application with only these permissions may not be vulnerable to PendingIntent related attacks.

Part IV

Related Work

State-of-the-Art Anti-theft Solutions

While the idea of a honeypot account for theft protection is novel, there has been extensive research in other theft protection mechanisms in Android. In case of a theft it is likely that a potential thief will remove the SIM card from the device and factory reset it. At the time of this writing, no existing anti-theft mechanism can protect the user's privacy, maintain his data and keep good chances that the device will be found again at the same time.

Dhanu et al. [75] created a software for theft protection. After the theft protection software was installed and configured, it waited for a replacement of the SIM card. Once the SIM card was replaced, it started collecting video material, location information, and sent them via MMS to a phone number previously configured by the device owner.

Also Ajay Shetty [74] proposed an anti theft software that was triggered by a replacement of the SIM card. Whenever this event occurred, the software sent a notification SMS to a preconfigured number. From that point of time, it was possible for the device owner to retrieve the current location of the device via a SMS request.

Chouhan et al. [16] created a theft protection software in the form of a web based remote administration tool. Over a web interface the device owner could request the current location of the device. The software also enabled the device owner to record voices of the thief, wipe his/her private data and read the web history from the thief. In addition, the software notified the device owner about replacements of the SIM card.

The work of Al Rassan and Al Sheikh [3] proposed an anti-theft system that was supported by SMS. After being activated by a specially crafted SMS, an application, that was previously installed on the stolen phone could either broadcast its location or lock personal data that was stored on the phone. This data included media and log files, as well as SMS and MMS records.

Kuppusamy et al. [51] proposed a system for theft protection that could also be used as simple remote administration tool. Via SMS messages it was possible to locate the device, erase critical data, trace calls, manage incoming SMS and system access.

Yu et al. [89] proposed a system for remotely wiping stolen phones. This system

worked in a way that a device owner could register his/her phone at the emergency call service provider. He could now from any point of time report the phone as stolen to the emergency call service provider. Additionally a background application was installed on the phone. Whenever the SIM card of the phone was removed, the application sent a wipe request to the emergency call service provider. If the device was stolen, the emergency call service provider answers this request with a modified call declined request, after which the phone would be wiped by the application. This scheme had the benefit that it neither requires a WIFI connection, nor an inserted SIM card to trigger the device wiping. However, the personal data and device was lost.

In all of the mentioned work, the thief has initial access to the user data until the anti-theft mechanism is triggered either automatically after a certain event occurs or manually by the device owner. The even more severe limitation of the mentioned approaches is that none of them prevents resetting the device. For this reason in each of this work, after the thief has triggered a factory reset, the anti-theft mechanisms are deleted and there are no chances that the device owner can regain his/her device.

Apart from academic work, there exist commercial solutions for locating and securing a lost or stolen device. Examples for these products are Apple's "Find My Device" [4], Avast's "Free Mobile Security" [6] or Symantec's "Norton Anti-Theft" [78]. These solutions can lock the device, locate it and provide additional functionality for mitigating damage in case of loss or theft. Additionally to the limitations mentioned previously, these solutions suffer from permission restrictions that are imposed by the Android Security Model. These restrictions lead to severe flaws. This insight is supported by the work of Simon and Anderson [76] that have examined various mobile anti-virus solutions for Android. They discovered failures in the implementation of the remote lock and wipe functionalities of these applications. Beside the restrictions imposed by the Android Security Model, they see the reasons for these flaws in certain vendor customizations.

Markus Schneider [72] developed a password manager that uses a similar approach for another domain. This password manager returns fake password information when a wrong master password is used.

Srinivasan and Wu [77] proposed a mechanism that primarily prevented the smartphone from being turned off or being silenced in case of a theft. This approach was

implemented by protecting the called functionality with passwords. Additionally, their proposed mechanism could wipe the device after a certain amount of failed password guesses. They rely on a password for preventing the thief accessing sensitive data. This measure is good for protecting the user's privacy but will at the same time trigger the thief to factory reset the device. After the factory reset, the anti theft mechanism will be deleted and there will be no chances for the owner to regain his/her device.

Another approach for protecting the privacy of user data in the case of a device theft was proposed by Tang et al. [80]. In their approach sensitive user data was encrypted and saved in the cloud. Their goal was to minimize the amount of sensitive data that was stored on the phone. The access to this cloud storage could now be restricted by any means, such as access rate limits, complete blocking as soon as the device was reported stolen and logging access. This approach focuses on protecting sensitive data of the user. Unfortunately, it could not help the owner regain his/her device.

State-of-the-Art ICC Analyzers

R-Droid [7] is an information flow analysis tool. It supports multi-threading and AsyncTasks and focuses on the resolution of strings for information flow purposes. As is, R-Droid does not support intents but conservatively reports every flow to an intent sender function as a leak.

Arzt et al. proposed Flowdroid [5], a static taint analysis tool that includes an extensive component lifecycle model. Flowdroid does not support certain callback methods and cannot analyse string manipulations. For these reasons its results are neither precise nor sound.

AppScan [43] is a commercial tool for detecting vulnerabilities in mobile and web applications, including information leaks in Android applications. However, it only supports intra-app inter-component communication analysis. Further, AppScan requires source code of the inspected application.

IccTA, proposed by Li et al. [53], leverages static taint analysis to analyze inter-component information flow leaks. However, IccTA neither handles multithreading nor complex string analysis. Further, IccTA ignores some "rarely used ICC

methods such as startActivities" [53], which leads to missed information leaks. Finally, IccTA requires the usage of APKCombiner [52], which does not scale to larger applications.

Klieber et al. proposed DidFail [50], a tool that analyzes information flow in Android applications. However, DidFail is limited to the analysis of Android Activities and implicit intents.

Wei et al. proposed Amandroid [84], which computes multiple graphs and uses them for resolving intents similarly to ordinary function calls. However, Amandroid ignores several sink functions, such as *startActivityForResult* and *bindService*.

Epicc, proposed by Octeau et al. [62] is a tool for the analysis of Android inter-component communication. Unlike IIFA, it focuses on ICC related vulnerabilities and does not consider data flows within an application. Jiang and Xuxian [49] proposed ContentScope, a dataflow analysis tool that detects integrity and confidentiality vulnerabilities. However, ContentScope is limited to only Content Providers. Gordon et al. [36] proposed DroidSafe, a tool for the detection of information flow vulnerabilities. DroidSafe does not analyze the lifecycle of Android applications leading to imprecision. DroidSafe's precision is further reduced by its missing flow-sensitivity. Li et al. [54] proposed PCLeaks, a data-flow analysis tool for the detection of information leaks and component hijacking in Android applications. PCLeaks cannot handle multithreaded programs and multiple ICC sink methods such as *startActivities*.

Hunang et al. [42] proposed a type system for the prevention of data disclosure. Unlike our approach, they require annotations in the source code, which puts an extra burden on application developers. Barros et al. [8] developed a type system for the resolution of intents and reflections. This resolution requires annotated source code. Additionally, they inherit the imperfections of their underlying framework Epicc [62]. ScanDroid is a tool proposed by Fuchs et al. [24] for the analysis of Android intents via a constraint system. In this approach, the lack of distinction between component contexts leads to imprecise analysis results.

DroidChecker, proposed by Chan et al. [13] is a taint analysis tool for data flows within Android applications. Because of imprecise permission handling, DroidChecker is neither sound, nor complete. Further it cannot handle dynamic features of Java, such as polymorphism. Enck et al. [21] proposed TaintDroid, a dynamic taint-analysis tool that supports intents. TaintDroid is especially suited for finding

leaks of sensitive data in vulnerable applications. As TaintDroid is a taint based approach, it cannot differentiate between different sources of sensitive data.

Octeau et al. [61] implemented IC3, an analysis tool for Android intents. Their approach requires a specification of the intended program behavior, written in a declarative language COAL. However, the creation of the COAL specification requires access to the source code and expert knowledge. Liu et al. [57] proposed MR-Droid, which generates an information flow graph and computes risk scores for various vulnerabilities. However, it does not natively support groups of more than two applications and misses various details for creating ICC links. DroidDisintegrator by Schuster et al. [73] applied dynamic analysis using a device emulator. This emulator monitors app component communication, generates policies and enforces them directly in the application. However, it is limited to intra-application information flow and is prone to false positives and false negatives.

State-of-the-Art PendingIntent Analyzers

To the best of our knowledge, there exists no approach that detects the described PendingIntent vulnerabilities at the time of this writing. Bugiel et al. [11] proposed XManDroid, a reference monitor to prevent privilege escalation attacks. Their approach is focused on application permissions and policies to model the desired application privileges. In contrast to our approach, XManDroid only regards PendingIntents as vehicle for inter-component communication and does not consider the peculiarities and vulnerabilities of PendingIntents.

SAAF [2], proposed by Hoffmann et al., is a tool to statically analyze SMALI code. It recovers String constants from backward slices of method calls in order to detect suspicious behavior. However, as it could not produce the expected results in our experiments with current APKs we re-implemented a SMALI slicer.

Li et al. [55] analyzed vulnerabilities in Google's GCM and Amazon's ADM mobile-cloud services. They discovered a critical logical flaws, concerning both of these services. Additionally, they discovered a PendingIntent vulnerability in GCM. Unlike our approach, they discovered this vulnerability by manual code analysis and they do not provide an automated approach for discovering PendingIntent vulnerabilities.

Apart from work considering PendingIntents, there is an extensive body of work on general Intent analysis. One line focuses on Intent fuzzers for finding Intent related vulnerabilities. For example, Yang et al. [87] developed an Intent fuzzer for the detection of capability leaking vulnerabilities in Android applications. JarJarBinks, proposed by Maji et al. [59], is a fuzzing tool for Android intents. By sending a large number of requests, the authors found robustness vulnerabilities in the Android runtime environment. Sasnauskas and Regher [71] created an Intent Fuzzer. Their approach is based on static analysis and generates random test-cases based on the analysis results. In contrast to our approach, their approaches do not consider PendingIntent related vulnerabilities.

Other work focuses on Intent based test case generation. For example, Jha et al. [48] proposed a model that abstracts Android inter-component communication. From this model the authors derived test cases that facilitates the software engineering process. Salva and Zafimiharisoa [70] proposed APSET, a tool that implements a model-based testing approach for Android applications. The proposed approach generates test cases that check for the leakage of sensitive information. In contrast, our approach is focused on the security perspective of PendingIntents.

As Intents are extensively used by malware, some approaches use Intent analysis as a feature for malware detection. Feizollah et al. [22] proposed AndroDialysis, a tool that uses Intents as indicating feature for Android malware. Tam et al. [79] proposed CopperDroid, a monitor system which tracks events via virtual machine introspection. CopperDroid considers PendingIntent as vehicle for Inter Process Communication. In contrast, our approach is intended for finding PendingIntent related vulnerabilities in benign applications.

Several approaches use various static information flow techniques for Intent analysis. Sadeghi et al. [69] proposed COVERT, a static analysis tool for the analysis of Intents. It computes information flows by static taint analysis. Using COVERT, the authors discovered hundreds of vulnerabilities in applications from the Google Play Store and other sources. Yang et al. [88] proposed AppIntent, an analysis tool for finding leakage of sensitive information via Intents. The key idea of their approach is to distinguish intended information leakage from unintended leakage considering user interface actions. The authors leverage the Android execution model to perform an efficient symbolic execution analysis. Unlike ours, both approaches do not consider PendingIntent related vulnerabilities.

Chin et al. [15] proposed ComDroid, a tool for detecting inter-component related vulnerabilities, e.g., Intent spoofing or Service Hijacking. Chan et al. [14] proposed an approach to detect privilege escalation attacks in Android applications. As their approach does not include any kind of information flow control, it overapproximates possible attacks leading to reduced precision. Again, both do not detect security vulnerabilities caused by PendingIntents.

Part V

Conclusion and Outlook

Chapter 6

Conclusion

The exponential growth in smartphones' usage has lured several malicious entities to hunt for users' sensitive data. While Android is the most favorable smartphone operating system, security measures to protect against dubious activities are limited. Modern state-of-the-art analyses took great effort to enhance these security measures. Present anti-theft frameworks provide great features in case of loss of the device , e.g, remote tracking of the device. Unfortunately, they do not work if they do not have an active network connection, e.g., if the SIM card was removed from the device. In addition, recent research considered static program analysis to detect potentially dangerous data leaks. Yet, state-of-the-art information flow analyses suffer from several shortcomings with respect to precision, soundness, and scalability. This thesis addresses these problems by:

- **Developing the framework to protect the users' privacy/data against physical device theft/loss:** In this work we propose ThiefTrap, a novel concept using a honeypot account for the purpose of theft protection. Using this concept it is possible to protect sensitive user data while retaining high chances to regain the device. Our novel approach is the first that can achieve this combination of desired properties. We implemented our approach as modifications on the latest version of the Android operating system, the most used operating system in mobile devices at the time of this writing. Based on this implementation we successfully evaluated our approach in an empirical user

study including 35 participants. The results of our study show that it is not possible for a user to distinguish the honeypot account from a regular unlocked device. Additionally, we could retrieve information of the participants that in a real world scenario could be used to regain the device. It should be mentioned that the proposed concept is universal and can be customized for various scenarios and platforms.

- **Developing the frameworks to protect the users' privacy/data against (un) intentional application leaks:** In this work we propose two novel frameworks, IIFA and PIAnalyzer, for the information flow analysis in Android applications.

  IIFA focuses on the information flow analysis of intents. In this work we developed a novel information-flow analysis for IAC (and ICC) based on an intent-flow pre-analysis that evades combinatorial explosion of analyzing all potential communication partners, while precisely matching type and key information of intent data as well as apps registered to receive certain implicit intents. We compared IIFA with six related tools on four standard benchmark sets. IIFA achieves 100% precision and recall rates on both benchmark sets and thereby significantly outperformed every previous tool. We finally applied IIFA on the 90 most popular applications from the Google Playstore and showed that in contrast to state-of-the-art tools the runtime performance of IIFA scales to large real-world applications.

  PIAnalyzer is the first approach to analyze and detect PendingIntent-related vulnerabilities. PIAnalyzer is fully automated and does neither require the source code of the applications under inspection, nor any effort by the analyst. We evaluated PIAnalyzer on 1000 randomly selected applications from the Google Play Store to assess the runtime performance, precision and soundness of PIAnalyzer. PIAnalyzer takes on average only approximately 13 seconds per application, which scaled up well to the large test sets. PIAnalyzer discovers 1288 warnings and 70 PendingIntent vulnerabilities. We manually investigated some of the reports and elaborated on a privilege escalation vulnerability caused by the usage of a prevalent Google library.

Chapter 7

Future Work

7.1 Protection for the Alternative Physical Storage Medium

This thesis provides ThiefTrap, an anti-theft framework for the theft/loss protection of an Android device. While ThiefTrap is an important step in the development of anti-theft mechanisms, it can be further extended in the future. Potential extensions include the protection of alternative storages of private user data, such as the SIM card and the device settings. In our approach, these storages were excluded, as the SIM card is rarely used today for storing personal information and the device settings do not contain highly sensitive information. Still, there are some users who would like also to protect these places. A further point of future work is the creation of alternative mechanisms that simulate the owner's account data from within the honeypot account. These mechanisms can include the automatic or semi-automatic generation of fake data.

7.2 Improving the String Analysis for ICC

IIFA models String and List APIs in order to recover strings created using concatenation, substring, etc. More contrived examples like converting a string to an array

of chars (to be manipulated) can be further extended in the future. Due to our modular design we could also add more expensive analyses like SMT-solvers to handle more cases.

7.3 Adding IFC for Android's Hybrid-App Communication

Hybrid mobile apps combine native app components with web app components into a single mobile application. Intuitively, hybrid apps are native applications combined with web technologies such as HTML, Javascript and CSS. In Android, such a communication is achieved via a *WebView*, which is a chromeless browser capable of displaying webpages. The developer survey from App Trends [47] shows the increasing prevalence of the hybrid applications: In the last two years (2015-17), app development with native tools has decreased significantly (nearly 7x decrease). Whereas, the number of hybrid-built apps is growing as a share of overall app development. By the end of 2019, 32.7% of developers surveyed expect to completely abandon native development in favor of hybrid.

Due to the fact that hybrid apps combine native and web technologies in a single app, the attack surface for malicious activities increases significantly as potentially untrusted code that is loaded at runtime can interfere with the trusted Android environment. In our study with 7,500 random applications from Google Play Store, we find that 68% of these apps use at least one instance of the WebView and 87.9% of these install an active communication channel between Android and JavaScript. These communications include exchange of various pieces of sensitive information, such as the user's location information. To assess the impact on user privacy a standalone analysis of the Android or Javascript side is thus clearly insufficient. A comprehensive solution comprise of IFC on combination of JavaScript and Android can be further extended in the future.

List of Figures

List of Tables

Appendix

A Smali Code for the Creation of a PendingIntent

```
.method protected onCreate(Landroid/os/Bundle;)V
    # 5 local registers are used in this method
    .locals 5

    # Declaration of a parameter register with a given name
    .param p1, "savedInstanceState" # Landroid/os/Bundle;

    # End of the method prologue. Start of the actual code.
    .prologue

    # A call to a super constructor
    invoke-super {p0, p1},
    Landroid/support/v7/app/AppCompatActivity;->
    onCreate(Landroid/os/Bundle;)V

    # Declaration of a String in register v3
    const-string v3, "android.intent.action.CALL"

    # Creation of an Intent object in register v0
    new-instance v0, Landroid/content/Intent;

    # Invocation of the constructor of the intent object
    invoke-direct {v0, v3},
    Landroid/content/Intent;-><init>(Ljava/lang/String;)V

    # Return of the method with no return value
    return-void

.end method
```

A.1 Related works Comparison

Table 1: Comparison results. ✓: true positive, ⋆: false positive, ○: false negative

IccTA								
Test Case	Explicit ICC	Flow Droid	App Scan	Did Fail	DIAL droid	Aman droid	IccTA	Our Tool
startActivity1	T	✓ ⋆	✓ ⋆	○	✓	✓	✓	✓
startActivity2	T	✓ (4 ⋆)	✓ (4 ⋆)	○	✓	✓	✓	✓
startActivity3	T	✓(32 ⋆)	✓(32 ⋆)	○	✓	✓	✓	✓
startActivity4	F	⋆ ⋆	⋆ ⋆	⋆				
startActivity5	F	⋆ ⋆	⋆ ⋆	⋆				
startActivity6	T	⋆ ⋆	⋆ ⋆		⋆	⋆		
startActivity7	T	⋆ ⋆	⋆ ⋆		⋆	⋆	⋆	
startActivityFR1	T	✓	✓	○	✓	✓	✓	✓
startActivityFR2	T	✓	○	○	✓	○	✓	✓
startActivityFR3	T	✓ ⋆	○	○	○	○ ⋆	✓	✓
startActivityFR4	T	✓ ✓ ⋆	✓ ○	○ ○	✓ ○	✓ ○ ⋆	✓ ✓	✓ ✓
startService1	T	✓ ⋆	✓ ⋆	○	✓	✓	✓	✓
startService2	T	✓ ⋆	✓ ⋆	○	✓	○	✓	✓
bindService1	T	✓ ⋆	✓ ⋆	○	✓	○	✓	✓
bindService2	T	○	○	○	○	○	✓	✓
bindService3	T	○	○	○	○	○	✓	✓
bindService4	T	✓ ⋆ ○	✓ ⋆ ○	○ ○	✓ ○	○ ○	✓ ✓	✓ ✓
sendBroadcast1	F	✓ ⋆	✓ ⋆	○	✓	✓	✓	✓
ICC-Bench								
Explicit1	T	✓	-	○	✓	✓	✓	✓
Implicit1	F	✓	-	✓	✓	✓	✓	✓
Implicit2	F	✓	-	✓	✓	✓	✓	✓
Implicit3	F	✓	-	✓	✓	✓	✓	✓
Implicit4	F	✓	-	✓	✓	✓	✓	✓
Implicit5	F	✓ ⋆	-	✓	✓	✓	✓	✓
Implicit6	F	✓	-	✓	✓	✓	✓	✓
DynRegister1	F	○	-	○	✓	✓	✓	✓
DynRegister2	F	○	-	○	✓	○	○	✓
Attribute-Mismatch-ICC-Benchmark								
GetPutMismatch	T	⋆	-	-	⋆	⋆	⋆	
KeyAmbiguity	T	⋆	-	-	⋆	⋆	⋆	
KeyMismatch1	T	⋆	-	-	⋆	⋆	⋆	
KeyMismatch2	T	⋆	-	-	⋆	⋆	⋆	
KeyMismatch3	T	⋆	-	-	⋆	⋆	⋆	
KeyMismatch4	T	⋆	-	-				
IACTestSetA	T	-	-	-	⋆	-	-	
Sum, Precision, Recall and F1								
✓, higher is better		20	10	6	20	15	24	25
⋆, lower is better		59	50	2	8	9	6	0
○, lower is better		5	6	19	5	10	1	0
Precision $p = ✓/(✓ + ⋆)$		25%	16.7%	75%	71%	62%	80%	100%
Recall $r = ✓/(✓ + ○)$		80%	62.5%	24%	80%	60%	96%	100%
F_1-measure $2pr/(p + r)$		0.41	0.26	0.36	0.75	0.61	0.87	1

Bibliography

[1] 2.0, A.: Apktool. GitHub (Jul 2017), https://ibotpeaches.github.io/Apktool/

[2] ACM (ed.): Slicing Droids: Program Slicing for Smali Code, vol. Proceedings of the ACM Symposium on Applied Computing. SAC, ACM, New York, NY, USA (2013)

[3] Al Rassan, I., Al Sheikh, M.A.: Securing application in mobile computing. International Journal of Information and Electronics Engineering 3(5), 544 (2013)

[4] Apple: Find my iphone, ipad, ipod touch, or mac. www.apple.com/support/icloud/find-my-device/

[5] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices 49(6), 259–269 (2014)

[6] Avast: Avast free mobile security (June 2017), http://www.avast.com/en-us/free-mobile-security

[7] Backes, M., Bugiel, S., Derr, E., Gerling, S., Hammer, C.: R-droid: Leveraging android app analysis with static slice optimization. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 129–140. ACM (2016)

[8] Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., Ernst, M.D., et al.: Static analysis of implicit control flow: Resolving java reflection and android intents (t). In: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on. pp. 669–679. IEEE (2015)

[9] Bosu, A., Liu, F., Yao, D.D., Wang, G.: 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, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017. pp. 71–85 (2017)

[10] Brains, J.: Kotlin, `https://kotlinlang.org`

[11] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)

[12] Cannon, T., Bradford, S.: Into the droid: Gaining access to android user data. In: DefCon Hacking Conference (DefCon'12), Las Vegas, Nevada, USA (2012)

[13] Chan, P.P., Hui, L.C., Yiu, S.M.: Droidchecker: analyzing android applications for capability leak. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 125–136. ACM (2012)

[14] Chan, P.P., Hui, L.C., Yiu, S.: A privilege escalation vulnerability checking system for android applications. In: Communication Technology (ICCT), 2011 IEEE 13th International Conference on. pp. 681–686. IEEE (2011)

[15] Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. pp. 239–252. ACM (2011)

[16] Chouhan, J.G., Singh, N.K., Modi, P.S., Jani, K.A., Joshi, B.N., et al.: Camera and voice control based location services and information security on android. Journal of Information Security 7(03), 195 (2016)

[17] Chris Klotzbach, D.a.F., Lali Kesiraju, M., at Flurry, A.M.: Flurry state of mobile 2017 (January 2018), `http://flurrymobile.tumblr.com/post/169545749110/state-of-mobile-2017-mobile-stagnates`

[18] Christian Fritz, S.A., Rasthofer, S.: Droid-benchmarks. `https://github.com/secure-software-engineering/DroidBench`, accessed: December. 2017

[19] CNBC: Cnbc study. http://www.cnbc.com/2014/04/26/most-americans-dont-secure-their-smartphones.html

[20] Dolby, J., Fink, S.J.: Wala framework. IBM T.J. Watson (June 2015), `http://wala.sourceforge.net/wiki/index.php/Main_Page`

[21] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32(2), 5 (2014)

[22] Feizollah, A., Anuar, N.B., Salleh, R., Suarez-Tangil, G., Furnell, S.: Androdialysis: Analysis of android intent effectiveness in malware detection. Computers & Security 65, 121–134 (2017)

[23] Freke, J.: Baksmali. https://github.com/JesusFreke/smali

[24] Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android. Tech. rep., University of Maryland (2009)

[25] Google: Android intent documentation. `https://developer.android.com/reference/android/content/Intent.html`, accessed: May. 2017

[26] Google: Dalvik byteycode documentation. `https://source.android.com/devices/tech/dalvik/dalvik-bytecode`, accessed: May. 2017

[27] Google: Pending intent documentation. `https://developer.android.com/reference/android/app/PendingIntent.html`

[28] Google: Android device manager (June 2017), `https://www.google.com/android/devicemanager`

[29] Google: Android device protection (June 2017), `https://support.google.com/nexus/answer/6172890`

[30] Google: Android internals (February 2017), `https://source.android.com/source/index.html`

[31] Google: Android open source project (June 2017), `https://source.android.com`

[32] Google: Android os version usages (January 2017), https://developer.android.com/about/dashboards/index.html

[33] Google: Android version usage (July 6 2017), https://developer.android.com/about/dashboards/index.html

[34] Google: Android os statistics (Feb 2018), https://developer.android.com/about/dashboards/index.html#Screens

[35] Google: Android permissions (April 2018), https://developer.android.com/guide/topics/permissions/overview.html

[36] Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS. Citeseer (2015)

[37] Grech, N., Kastrinis, G., Smaragdakis, Y.: Efficient Reflection String Analysis via Graph Coloring. In: Millstein, T. (ed.) 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 109, pp. 26:1–26:25. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2018), http://drops.dagstuhl.de/opus/volltexte/2018/9231

[38] Groß, S., Tiwari, A., Hammer, C.: Pianalyzer: A precise approach for pending-intent vulnerability analysis. In: Lopez, J., Zhou, J., Soriano, M. (eds.) Computer Security. pp. 41–59. Springer International Publishing, Cham (2018)

[39] Groß, S., Tiwari, A., Hammer, C.: Thieftrap – an anti-theft framework for android. In: Security and Privacy in Communication Networks. pp. 167–184. Springer International Publishing, Cham (2018)

[40] GS: Android global market share. http://gs.statcounter.com/os-market-share/mobile/worldwide (December 2018)

[41] Höschler, E.: Information flow control for Hybrid Android applications. Bachelor thesis, University of Potsdam (2018)

[42] Huang, J., Zhang, X., Tan, L.: Detecting sensitive data disclosure via bi-directional text correlation analysis. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 169–180. ACM (2016)

[43] IBM: Ibm security appscan source. https://www-03.ibm.com/software/products/en/appscan, accessed: May. 2017

[44] IDC: Worldwide smartphone os market share (Nov 2016), http://www.idc.com/promo/smartphone-market-share/os;jsessionid=6A0934D1434A49DBFFE74D63DA2C595B

[45] Insider, B.: Idg research (May 2014), http://www.businessinsider.com/smartphone-theft-statistics-2014-5?IR=T

[46] Insight, S.: Statistics on consumer mobile usage (May 2018), https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

[47] Ionic: Developer survey (2018), https://ionicframework.com/survey/2017#trends

[48] Jha, A.K., Lee, S., Lee, W.J.: Modeling and test case generation of inter-component communication in android. In: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems. pp. 113–116. IEEE Press (2015)

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

[50] Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: 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. ACM (2014)

[51] K.S. Kuppusamy, Senthilraja. R and G. Aghila: A model for remote access and protection of smartphones using short message service. arXiv preprint arXiv:1203.3431 (2012)

[52] Li, L.: Apk combiner. GitHub (December 2014), https://github.com/lilicoding/ApkCombiner

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

[54] Li, L., Bartel, A., Klein, J., Le Traon, Y.: Automatically exploiting potential component leaks in android applications. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on. pp. 388–397. IEEE (2014)

[55] Li, T., Zhou, X., Xing, L., Lee, Y., Naveed, M., Wang, X., Han, X.: Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 978–989. ACM (2014)

[56] Limited, G.G.: Table tennis 3d. Google Play store (April 2014)

[57] Liu, F., Cai, H., Wang, G., Yao, D.D., Elish, K.O., Ryder, B.G.: Mr-droid: A scalable and prioritized analysis of inter-app communication risks. Proc. of MoST (2017)

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

[59] Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in android. In: Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on. pp. 1–12. IEEE (2012)

[60] Oberheide, J., Miller, C.: Dissecting the android bouncer. SummerCon2012, New York (2012)

[61] Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P.: Composite constant propagation: Application to android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 77–88. IEEE Press (2015)

[62] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX security symposium. pp. 543–558 (2013)

[63] Ossmann, M., Osborn, K.: Multiplexed wired attack surfaces. BlackHat USA (2013)

[64] Pauck, F., Bodden, E., Wehrheim, H.: 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. ESEC/FSE 2018, ACM, New York, NY, USA (2018)

[65] Qiu, L., Wang, Y., Rubin, J.: 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. ACM (2018)

[66] Qiu, L., Wang, Y., Rubin, J.: 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. ISSTA 2018 (2018)

[67] Reports, C.: Consumer reports (May 2014), `http://www.consumerreports.org/cro/news/2014/04/smart-phone-thefts-rose-to-3-1-million-last-year/index.htm`

[68] Ruddock, D.: Anti-theft. http://www.androidpolice.com/2015/03/12/guide-what-is-android-5-1s-antitheft-device-protection-feature-and-how-do-i-use-it/

[69] Sadeghi, A., Bagheri, H., Malek, S.: Analysis of android inter-app security vulnerabilities using covert. In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on. vol. 2, pp. 725–728. IEEE (2015)

[70] Salva, S., Zafimiharisoa, S.R.: Data vulnerability detection by security testing for android applications. In: Information Security for South Africa, 2013. pp. 1–8. IEEE (2013)

[71] Sasnauskas, R., Regehr, J.: Intent fuzzer: crafting intents of death. In: Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA). pp. 1–5. ACM (2014)

[72] Schneider, D.M.: Imobilesitter. http://www.imobilesitter.com/ (March 2014)

[73] Schuster, R., Tromer, E.: Droiddisintegrator: intra-application information flow control in android apps. In: ACM Asia Conference on Computer and Communications Security (ASIACCS) to appear (2016)

[74] Shetty, A.: Mobile anti theft system (mats). (2012)

[75] Shweta Dhanu, Afsana Shaikh, S.B.: Anti-theft application for android based devices. International Journal of Advanced Research in Computer and Communication Engineering (2016)

[76] Simon, L., Anderson, R.: Security analysis of consumer-grade anti-theft solutions provided by android mobile anti-virus apps. In: 4th Mobile Security Technologies Workshop (MoST). Citeseer (2015)

[77] Srinivasan, A., Wu, J.: Safecode–safeguarding security and privacy of user data on stolen ios devices. In: Cyberspace Safety and Security, pp. 11–20. Springer (2012)

[78] Symantec: Norton mobile security, `https://us.norton.com/anti-theft/`

[79] Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: Copperdroid: Automatic reconstruction of android malware behaviors. In: NDSS (2015)

[80] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., Sarda, N.: Cleanos: limiting mobile data exposure with idle eviction. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 77–91 (2012)

[81] Tiwari, A.: Joana framework with dexlib2. GitHub (2017), `https://gitup.uni-potsdam.de/tiwari/Joana-dexlib-2`

[82] Tiwari, A., Groß, S., Hammer, C.: Iifa – a modular inter-app intent information flow analysis of android applications (2018), https://arxiv.org/abs/1812.05380v1

[83] WangTao, Zhang Donghui, W.: Android settings pendingintent leak. https://packetstormsecurity.com/files/129281/Android-Settings-Pendingintent-Leak.html (November 2014)

[84] Wei, F., Roy, S., Ou, X., 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)

[85] Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (Jul 1984)

[86] Whitwam, R.: Anti-theft (2015), http://www.greenbot.com/article/2904397/everything-you-need-to-know-about-device-protection-in-android-51.html

[87] Yang, K., Zhuge, J., Wang, Y., Zhou, L., Duan, H.: Intentfuzzer: detecting capability leaks of android applications. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 531–536. ACM (2014)

[88] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1043–1054. ACM (2013)

[89] Yu, X., Wang, Z., Sun, K., Zhu, W.T., Gao, N., Jing, J.: Remotely wiping sensitive data on stolen smartphones. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 537–542. ACM (2014)