Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Understanding and Assessing Security on Android via Static Code Analysis

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von

Erik Derr

Saarbrücken,
December 2017

Erik Derr
Understanding and Assessing Security on Android via Static Code Analysis

Tag des Kolloquiums: 28.08.2018

Dekan: Prof. Dr. Sebastian Hack

Prüfungsausschuss:

Vorsitzender: Prof. Dr. Christian Rossow

Berichterstattende: Prof. Dr. Michael Backes
Prof. Dr. Andreas Zeller
Prof. Patrick McDaniel, Ph.D.

Akademischer Mitarbeiter: Dr. Yang Zhang

Abstract

Smart devices have become a rich source of sensitive information including personal data (contacts and account data) and context information like GPS data that is continuously aggregated by onboard sensors. As a consequence, mobile platforms have become a prime target for malicious and over-curious applications. The growing complexity and the quickly rising number of mobile apps have further reinforced the demand for comprehensive application security vetting.

This dissertation presents a line of work that advances security testing on Android via static code analysis. In the first part of this dissertation, we build an analysis framework that statically models the complex runtime behavior of apps and Android's application framework (on which apps are built upon) to extract privacy and security-relevant data-flows. We provide the first classification of Android's protected resources within the framework and generate precise API-to-permission mappings that excel over prior work. We then propose a third-party library detector for apps that is resilient against common code obfuscations to measure the outdatedness of libraries in apps and to attribute vulnerabilities to the correct software component. Based on these results, we identify root causes of app developers not updating their dependencies and propose actionable items to remedy the current status quo. Finally, we measure to which extent libraries can be updated automatically without modifying the application code.

Zusammenfassung

Smart Devices haben sich zu Quellen persönlicher Daten (z.B. Kontaktdaten) und Kontextinformationen (z.B. GPS Daten), die kontinuierlich über Sensoren gesammelt werden, entwickelt. Aufgrund dessen sind mobile Platformen ein attraktives Ziel für Schadsoftware geworden. Die stetig steigende App Komplexität und Anzahl verfügbarer Apps haben zusätzlich ein Bedürfnis für gründliche Sicherheitsüberprüfungen von Applikationen geschaffen.

Diese Dissertation präsentiert eine Reihe von Forschungsarbeiten, die Sicherheitsbewertungen auf Android durch statische Code Analyse ermöglicht. Zunächst wurde ein Analyseframework gebaut, dass das komplexe Laufzeitverhalten von Apps und Android's Applikationsframework (dessen Funktionalität Apps nutzen) statisch modelliert, um sicherheitsrelevante Datenflüsse zu extrahieren. Zudem ermöglicht diese Arbeit eine Klassifizierung geschützter Framework Funktionalität und das Generieren präziser Mappings von APIs-auf-Berechtigungen. Eine Folgearbeit stellt eine obfuskierungs-resistente Technik zur Erkennung von Softwarekomponenten innerhalb der App vor, um die Aktualität der Komponenten und, im Falle von Sicherheitlücken, den Urheber zu identifizieren. Darauf aufbauend wurde Ursachenforschung betrieben, um herauszufinden wieso App Entwickler Komponenten nicht aktualisieren und wie man diese Situation verbessern könnte. Abschließend wurde untersucht bis zu welchem Grad man veraltete Komponenten innerhalb der App automatisch aktualisieren kann.

Background of this Dissertation

This dissertation is based on the papers mentioned in the following. I contributed to all papers as the main author.

The initial work [T1, P1] is based on the author's master's thesis proposed by Sebastian Gerling. Sven Bugiel, Sebastian Gerling and Christian Hammer were involved in general writing tasks. All authors performed reviews of the paper.

The idea for the work on Android's middleware [P2] was developed on a visit of Patrick McDaniel and Damien Octeau at the Pennsylvania State University in December 2014. Damien Octeau and Sven Bugiel contributed with their in-depth knowledge about framework internals and in writing the paper. Sebastian Weisgerber contributed to the implementation of our prototype. All authors performed reviews of the paper.

The idea for LibScout [P3] originates from the author's previous experience on app analysis [T1, P1]. Sven Bugiel contributed by establishing an app database with complete version histories. He was further involved in the process of writing the paper. All authors performed reviews of the paper.

Based on the results of LibScout [P3], the follow-up paper [P4] dealt with the question of library updatability. The idea for the app developer survey was discussed between the author, Sven Bugiel, Sascha Fahl and Yasemin Acar. Parts of the app developer survey were conducted by Sascha Fahl and Yasemin Acar. Besides general discussions, Sven Bugiel was further involved in writing tasks. All authors performed reviews of the paper.

Author's Papers for this Thesis

[P1] Backes, M., Bugiel, S., Derr, E., Gerling, S., and Hammer, C. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS 2016)*. ACM, 2016.

[P2] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: *Proceedings of the 25th USENIX Security Symposium (SEC 2016)*. USENIX Association, 2016.

[P3] Backes, M., Bugiel, S., and Derr, E. Reliable Third-Party Library Detection in Android and its Security Applications. In: *Proceedings of the 23rd ACM Conference on Computer and Communication Security (CCS 2016)*. ACM, 2016.

[P4] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In: *Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS 2017)*. ACM, 2017.

Further Publications of the Author

[S1] Bugiel, S., Derr, E., Gerling, S., and Hammer, C. Advances in Mobile Security. In: *8th Future Security - Security Research Conference*. Fraunhofer Verlag, 2013.

[S2] Derr, E. The Impact of Third-party Code on Android App Security. *Usenix Enigma (Enigma 2018)* (2018).

[S3] Oltrogge, M., Derr, E., Stransky, C., Acar, Y., Fahl, S., Rossow, C., Pellegrino, G., Bugiel, S., and Backes, M. The Rise of the Citizen Developer: Assessing the Security Impact of Online Application Generators. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P 2018)*. IEEE, 2018.

Technical Reports of the Author

[T1] Backes, M., Bugiel, S., Derr, E., and Hammer, C. *Taking Android App Vetting to the Next Level with Path-sensitive Value Analysis*. Tech. rep. A/02/2014. Center for IT-Security, Privacy and Accountability (CISPA), Saarbrücken, 2014.

Acknowledgments

Pursuing a Ph.D. over a period of five years requires a lot of will and dedication and typically does not go without minor sacrifices in your personal life. Therefore having people that support you in this long and challenging journey is of utmost importance. In the following, I want to thank the people without whom this thesis would not have been possible.

First of all, I would like to thank my supervisor Michael Backes who convinced me to stay in academia after my master's thesis and who gave me the opportunity to work both in academia and industry. Throughout the years, he taught me the necessary knowledge to survive in the world of scientific research. In particular, I would like to thank him for his support to make my first publication possible after an emotionally and physically draining number of (re-)submissions. Thanks to Sebastian Gerling for proposing my master's thesis topic that caught my interest in mobile security and that eventually led to my dissertation topic. Special thanks to Philipp von Styp-Rekowsky—my longtime office mate—for the great time at work with a lot of fruitful discussions and to Sven Bugiel, for contributing to many of my scientific works and giving valuable feedback and advice. We soon also became friends and had a lot of fun at parties after work and on occasions such as karting, watching movies, or playing laser-tag. We liked traveling around the world to conferences and enjoyed to explore new places. In particular, our great trips to San Francisco and Washington, D.C. will stay in memory. In addition, I'm pleased to have had the chance to work with many talented and experienced collaborators and peers (in alphabetical order): Sascha Fahl, Christian Hammer, Jie Huang, Patrick McDaniel, Damien Octeau, Marten Oltrogge, Giancarlo Pellegrino, Christian Rossow, Oliver Schranz, Christian Stransky, and Sebastian Weisgerber. Finally, there are many more colleagues from the InfSec group and CISPA to thank for creating a great working environment and an enjoyable experience at work and beyond.

Last but not least, I want to sincerely thank my family. My parents have always given me the freedom and opportunity to make my own decisions. I'm very thankful that they have always supported me in any possible way. The same also applies to my wife Esther who often had to forego common time when deadlines were approaching or when I had to travel to a conference. In difficult times, she has always encouraged me to trust in my capabilities and to overcome the obstacles of scientific publishing upon negative notifications. Since beginning of 2016 we are a proud family with our son Elias who gave me new motivation and energy to go through with the dissertation. Eventually, my first paper was published shortly after his birthday and a number of successful submissions followed ever since.

Contents

List of Figures

List of Tables

List of Code Listings

1

Introduction

The advent of smart handheld devices has revolutionized our way of communicating with people. Traditional information sharing via short message services has been superseded by modern instant messengers and social platforms allow to disseminate news across the globe in a matter of seconds. Smart devices offer a plethora of features such as calendar and contacts management or video and photo collections that have been exclusive to desktop computers for years. In addition, they are equipped with a variety of sensors that extend the devices' capabilities such as GPS modules, gyroscopes, and even fingerprint sensors that greatly enhance user experience. Users can download third-party applications (or apps for short) from centralized marketplaces to add new functionality. In 2009, Apple coined the catchphrase "*There's an app for that*" to advertise that there is an application for (almost) any use case. This particularly holds true for Google's Play Store—the most prominent one for the Android ecosystem—that, according to market research, exceeded the number of 3 million hosted apps in 2017. The variety of apps includes banking apps, social media apps, games, finances, or utility apps. The success of this new app paradigm has also motivated the deployment of mobile software stacks to other platforms such as wearables, televisions, Internet of Things (IoT) and even cars. It has also changed the way of developing applications. Android apps are API-driven and programmed against software development kits (SDKs) that abstract from the complexity of the underlying framework services and provide simple means to interact with the system and other applications. Developers commonly re-use existing code in form of third-party libraries to add new functionality. This eases the development task but, at the same time, raises code complexity as library code is statically linked during app generation.

In consequence of its quickly growing popularity, Android has become a prime target for malware and overly curious applications. The rich source of private and sensitive data has fueled information-stealing malware and ransomware that holds the system captive by encrypting (user) data or locking the device and demanding a ransom. This motivated a line of security research that warned for such security and privacy risks and highlighted the imminent risk of libraries that inherit the set of permissions from their host apps. Due to code complexity and the binary-only distribution of apps, early code analysis approaches applied lightweight techniques to investigate apps' usage of permissions that are required to perform sensitive operations. More advanced static analysis techniques soon proved to be effective to extract security features such as sensitive data-flow information at scale, but coarse approximations of apps' runtime behavior led to imprecise results and to a high number of false alarms. Further, the scarce Android API documentation in terms of permission requirements motivated researchers to analyze the application framework to automatically generate comprehensive API-to-permission mappings. However, insufficient knowledge about the framework internals and lightweight analysis techniques to cope with the framework complexity resulted in inaccurate mappings that negatively influenced any application analysis that built upon these datasets. Despite their shortcomings, these approaches discovered new attack vectors and raised the security awareness of app developers and end-users. In particular, incidents of sensitive data leakage by commonly used advertisement and tracking libraries emphasized the potential for misuse by (closed-source) third-party code. However, the absence of a reliable library detector for application binaries impeded an accurate

identification of re-used code and thereby a correct attribution of vulnerabilities. As a consequence, existing analysis approaches tend to report over-approximating per-app statements that don't allow to effectively blame the original code author, i.e. the app developer or library developer.

The goal of this dissertation is to provide a static analysis framework that can provide insights into apps and the application framework and to extract security and privacy relevant data-flows. To this end, we improve on prior work in multiple dimensions. As a foundation for subsequent code analyses, we first generate static runtime models that mimic Android's complex event-driven lifecycle. Contrarily to prior work, we are then going to post-process the results of our data-flow analysis. Applying a set of static optimizations yields shorter and more expressive traces by re-assembling runtime values that will also significantly reduce the number of false alarms. The overall idea is that these results can be readily used for common analysis tasks such as privacy leak detection or API usage analysis. We like to further address the important, but widely disregarded, aspect of accountability by identifying commonly used third-party libraries in applications. This allows security vulnerabilities to be properly attributed to the correct principal, i.e. the app or library developer. A particular challenge constitutes the common use of bytecode obfuscation and dead-code elimination that makes the detection of exact library versions inherently difficult. Using our approach, we are going to empirically determine the outdatedness of third-party libraries in Google Play apps and analyze the time that it takes for app developers to adopt new library versions. We will test to which extent it is possible to automatically update library versions based on their API usage within apps. Finally, we seek to answer the question about root causes of app developers' abstinence of library updates and to give informed advice for remediation that involves the different actors in the Android ecosystem. On the middleware, the major challenge is to overcome the lack of a comprehensive knowledge base on how to statically analyze the application framework. Once established, the goal is to comprehensively investigate Android's permission system and to derive a high-level taxonomy of Android's protected resources, i.e. sensitive operations protected by permissions. Based on our gained knowledge, we will further provide novel API-to-permission mappings that eliminate the shortcomings of prior work.

Thesis Statement

This dissertation confirms the thesis that

> It is possible to create a framework that statically models the runtime behavior of Android apps and the Android application framework to extract privacy and security-relevant data-flows. In case of applications, these flows can be attributed to the correct entity, i.e. app code or library code. It is further possible to determine whether included libraries are outdated and, if so, to which extent they can be upgraded to newer versions automatically.

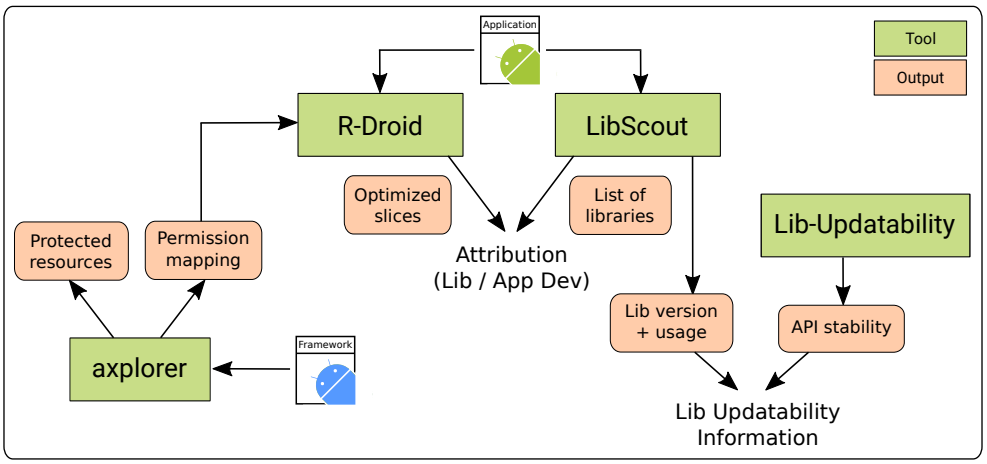


Figure 1.1: High-level overview of the analysis tools developed (shown in green), their relationship, and the respective output per tool (shown in orange).

Summary of Contributions

In the following, we summarize the major contributions of this dissertation and describe how the individual parts relate to each other. Figure 1.1 depicts a high-level overview of the analysis tools developed, their relationship, and the output each tool produces.

R-Droid R-Droid constitutes a comprehensive, general-purpose, static analysis framework for Android applications (cf. Chapter 3). It builds static runtime models that faithfully mimic the application lifecycle behavior. Building on top of these models, R-Droid employs a slicing-based analysis to extract data-dependent statements for arbitrary points of interest in an application. Prior work typically performs security assessments at this point, accepting that such traces may grow very large and contain spurious dependencies that lead to false positives. Our approach significantly improves on this problem by post-processing the results with a multi-stage optimization pipeline. The application of these optimizations—inspired by compiler optimization techniques—results in semantically equivalent, but significantly smaller slices with more expressive statements. This facilitates the task of understanding analysis results and enables automatic security assessments for a larger number of use-cases than prior work, e.g. analysis of arguments passed to security and privacy-sensitive APIs. R-Droid excels over related work in standard benchmarks in terms of precision and shows similar recall as the best-performing tools. To show its real-world applicability, we instantiated two common analysis tasks—data leakage detection and user input propagation analysis—as modules and found a significantly larger amount of privacy leaks than related tools when applied to apps from Google Play.

axplorer axplorer is an analysis framework to study the internals of the Android application framework (cf. Chapter 4). It is the first work to establish a solid and

comprehensive methodology on how to conduct a static analysis on the application framework. This particularly comprises the enumeration of the framework's public API and how to statically model the runtime behaviour in presence of design pattern that significantly differ from the application layer. Building on this knowledge base, we study Android's permission system, thereby answering *what* is actually protected by permissions. To this end, we identify the framework's protected resources, i.e. the privacy and security-sensitive operations that are guarded by Android's permissions, and establish a high-level taxonomy to classify them. Further, we re-visit the important use-case of mapping permissions to framework/SDK API methods. In particular, our novel mappings significantly improve on prior results that were based on insufficient knowledge about the framework's internals. Finally, we show that, although framework services follow the principle of separation of duty, the accompanying permission checks to guard sensitive operations violate it.

Accurate permission mappings are not only relevant for app developers due to an incomplete documentation, they also constitute a valuable source of sensitive APIs for application analysis, e.g. R-Droid may be configured to check for permission-protected APIs that retrieve sensitive information, such as location data or unique identifiers.

LibScout LibScout is a third-party library detector for Android applications that is resilient against common code obfuscation techniques (cf. Chapter 5). Our approach is capable of pinpointing exact library versions used in apps and, in case dead code elimination has been applied, to compute a similarity score to determine the fraction of library code included. Libraries are detected with profiles extracted from original library SDKs. This technique allows us to study library evolution in applications and to reliably identify library versions with known security vulnerabilities. This work was first to identify and quantify library outdatedness in apps of the Google Play store.

Being able to reliably distinguish app developer code from library code is highly relevant for various app analysis tasks. Identified privacy leaks or API misuse, as implemented in R-Droid, can then be properly attributed to the correct software component.

Library Updatability This work builds on top of LibScout and explores the problem space of *why* app developers do not update third-party libraries (cf. Chapter 6). A survey with app developers from Google Play provides first-hand information on library usage and requirements for more effective library updates. A subsequent study of library provider's semantic versioning practices uncovers that providers are likely a contributing factor to the app developers' abstinence from library updates in order to avoid ostensible re-integration efforts and version incompatibilities. In a large-scale library updatability analysis with apps from Google Play, we show that the vast majority of libraries could be upgraded by at least one version without modifying the app code. This also holds for library versions with known security vulnerabilities that could automatically be upgraded to the fixed version. Including the developer responses and the empirical results of our code studies, we give informed advice on how to remedy the current situation and propose actionable items for different actors in the Android ecosystem.

Outline

The remainder of this dissertation is structured as follows. Chapter 2 provides necessary background information on the Android software stack and on static analysis terminology. We present R-Droid in Chapter 3, axplorer in Chapter 4, and LibScout in Chapter 5. Chapter 6 presents an empirical study of third-party library updatability. We conclude this dissertation in Chapter 7.

2

Technical Background

This chapter introduces background information on the Android operating system and basics of program analysis techniques and data models that are used throughout chapters 3 to 6.

2.1 The Android Software Stack

Android is an open-source software stack for embedded devices that was released by Google in 2008. In the same year, Google founded the *Open Handset Alliance*—a consortium of hardware, software, and telecommunication companies—to develop and advance open standards for mobile devices. Since then, Android has gone through many major releases up to the most current release 8.0 (code name *Oreo*) in August 2017. Over the years, Android has become the most dominant smartphone operating system with a market share of about 81% by the end of 2016 [58]. Its major source of applications, or apps for short, is the *Play Store* operated by Google. In June 2017, the number of available apps passed 3 million [127] and users downloaded more than 65 billion apps within a twelve month period [138]. In contrast to Apple's iOS, third-party stores such as *Amazon's Appstore* exist as alternative marketplaces. Beyond being a platform for smartphones and tablets, Google has developed variations of Android for new use cases such as Android Wear (wearables) [15], Android TV (televisions) [14], Android Auto (cars) [13] and Android Things for the emerging market of IoT devices [46].

Linux Kernel The Android software stack comprises six layers (cf. Figure 2.1) with a modified Linux kernel as the foundation of the platform. The kernel has been adapted to the requirements of resource-constraint mobile devices with a specialized memory and power management. Further, Google modified the Binder Inter-process Communication (IPC) mechanism to allow seamless cross process boundary calls. In the second half of 2017, the majority of devices still runs on Linux 3.18 with long-term support that ended in January 2017. Starting with Android Oreo, Google enforces kernel version 4.4 to be used by device makers and introduces project *Treble* to facilitate platform updates for manufactures. In a nutshell, it provides a vendor interface to separate the vendor implementation—typically device-specific, low-level software—from the Android middleware [44].

Native Libs and Runtime On top of the Linux Kernel the extensive Android Middleware is built comprising core system components, the Runtime, and the application framework. System components such as the *Hardware Abstraction Layer* (HAL) or the Runtime ART are built from native code written in C and C++ and require native libraries such as *WebKit* or *OpenGL*. Prior to Android version 5.0, the *Dalvik* virtual machine (DVM) was the default runtime where each app runs in its own process and its own instance of the DVM. The Dalvik virtual machine is a register-based VM running dex (Dalvik executable) bytecode optimized for a low-memory footprint. Starting from Android 5.0 the DVM was superseded by the Android Runtime (ART) that comes with an on-device compiler to translate dex bytecode into highly optimized platform-specific

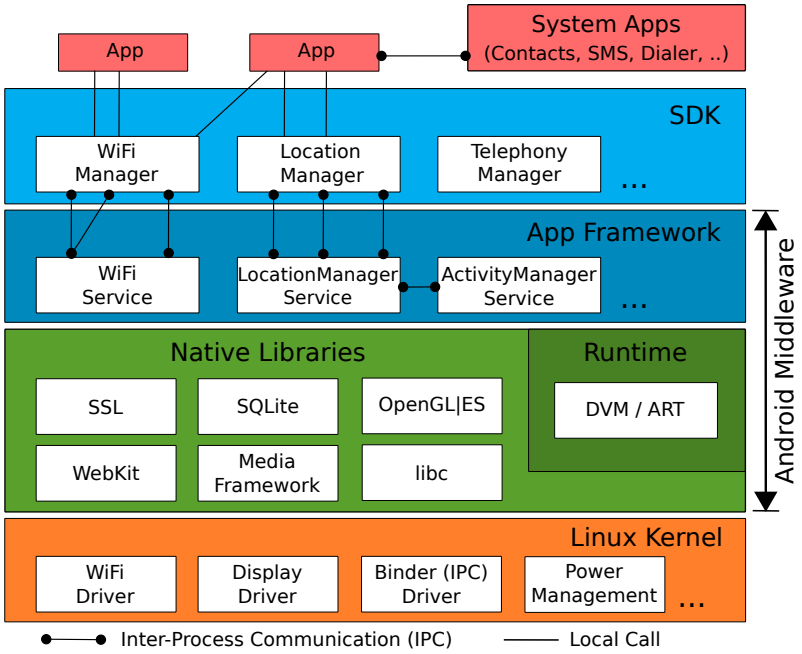


Figure 2.1: Android Software Stack with six different layers. Some IPC and local calls are depicted to exemplify different communication pattern across layers.

native code. Initially, the compilation process was done ahead-of-time during app installation. With Android 7.0, a just-in-time compiler was added to minimize compilation overhead.

Application Framework

The application framework comprises core system services written in Java and provides an API to access functionality such as retrieving location data or modifying the audio settings. Every framework service is responsible for providing access to one specific system resource, such as geolocation or radio interface. Some services utilize the *Java Native Interface* (JNI) to interact with the underlying platform through native components and libraries. For instance, the WifiService interacts with the WiFi daemon. Other services, such as Clipboard, do not rely on any hardware features.

Framework services are implemented as bound services [5] as part of the SystemServer. Bound service is the fundamental pattern to realize Android services that are remotely callable via a well-defined interface. Such interfaces are described in the *Android Interface Definition Language* (AIDL) and an AIDL compiler allows automated generation of Stub and Proxy classes that implement the interface-specific Binder-based RPC protocol to call the service. Here, Stubs are abstract classes that implement the Binder interface and need to be extended by the actual service implementation. Proxies are used by clients to call the service.

A small number of framework services does not use AIDL to auto-generate their Stub~/Proxy, but instead provides a custom class that implements the Binder interface. The

most prominent exception is the ActivityManagerService (AMS), which provides essential services such as application life-cycle management or Intent distribution. Since its interface is also called from native code—for which the AIDL compiler does not auto-generate native Proxies/Stubs and hence requires manual implementation of those—the RPC protocol for the AMS is hardcoded to avoid misalignment between manually written and auto-generated code.

Android SDK On top of Stubs and Proxies, the Android SDK provides Managers as an abstraction from the low-level RPC protocol. Manager classes encapsulate pre-compiled Proxies and allow developers to work with Manager objects that translate local method calls into remote-procedure calls to their associated service and hence enable app developers to easily engage into RPC with the framework's services. However, Proxies and Managers are just abstractions for the sake of app developer's convenience and nothing prevents an app developer from bypassing the Managers or re-implementing the default RPC protocol to directly communicate with the services.

System Apps System apps, such as Contacts, Dialer, or SMS complement the application framework with commonly requested functionality. However, in contrast to the application framework services that are fixed parts of any Android deployment, system apps are exchangeable or omittable (as can be observed in the various vendor customized firmwares) and, more importantly, are simply apps that are programmed against the same application framework API as third-party applications.

2.1.1 Permissions

One cornerstone of the Android security design are *permissions*, which an app must hold to successfully access the security and privacy critical methods of the application framework. Every application on Android executes under a distinct Linux UID and permissions are simply Strings[1] that are associated with the application's UID. There is no centralized policy for checking permissions on calls to the framework API. Instead, framework services that provide security or privacy critical methods to applications (must) enforce the corresponding, hard-coded permission that is associated with the system resources that the services expose. To enforce permissions, the services programmatically query the system whether their currently calling app—identified by its UID—holds the required permission and if not take appropriate actions (such as throwing an exception). For instance, the LocationManagerService would query the system whether a calling UID is associated with the String `android.permission.ACCESS_FINE_LOCATION`, which represents the permission to retrieve the GPS location data.

In this model, system apps differ from third-party apps in that they can successfully request security and privacy critical system permissions from the framework, which

[1] Permissions that map to Linux GIDs do not involve the framework and are not further considered here.

are not available to non-system apps. Moreover, like framework services (and any non-system application), they are responsible for enforcing permissions for resources they manage and expose on their RPC interfaces (e.g., contacts information or initiating phone calls). The difference to non-system applications is, that they usually enforce well-known permissions defined in the Android SDK, although the Android design does—in contrast to the framework services—not hardcode where those permissions are enforced, thus allowing system apps to be exchanged.

2.1.2 Android Apps

Application layer apps are written in Java and/or Kotlin [45] and are subsequently compiled to dex bytecode. On the device, this bytecode is either directly executed by the DVM or compiled to native code to be executed with ART. In addition to bytecode, app developers may also write code in C and C++ via the *Native Development Kit* (NDK). Similar to other software development platforms, it is common practice to integrate and re-use third-party components to facilitate app development. There are third-party libraries for (almost) any use case available ranging from advertisement libraries to monetize the application, over Android UI widgets, to utility libraries for XML/JSON processing. Libraries can be provided both as bundled Java bytecode (.jar file) or as an Android Archive Library (.aar), whereas the latter one provides resources in addition to bytecode. There is no dedicated package manager or centralized library store for Android. Libraries can be retrieved via various different release channels such as *Maven Central*, *JCenter*, or *GitHub*.

Application Build During the application build process, all code resources are compiled to one or more dex bytecode files. Multiple bytecode files are necessary when the number of declared methods exceeds the limit of 64K (a restriction of the dex bytecode format, typically referred to as 64K reference problem [43]). This implies that any third-party code is statically linked, causing large bytecode files in case of many dependencies. To remedy this problem, Android's developer IDE has built-in support for code minification. Internally, the widely used Java obfuscation tool *ProGuard*[68] is used to perform identifier renaming and dead code elimination to shrink and optimize the bytecode.

- **Dead code elimination** In this shrinking step unused code is identified and removed from the bytecode file. Typically a (conservative) reachability analysis is conducted from a set of predefined entry points to determine the set of classes and class members in use. Any other code is discarded. This is particularly effective for third-party components as app developers typically use only a subset of the library API.
- **Identifier renaming** is a technique to rename classes and class members that are not entry points. Original names are transformed into short, non-meaningful identifiers, i.e. a package name com.facebook might be renamed to a.b . Identifiers are still unique within the app, but these shortened names make any reverse engineering attempt more difficult. Further, this approach does also shrink the

overall app size, as the new identifiers are typically (much) smaller than the original ones.

These two shrinking/obfuscation techniques are side-effect free and have a low (one-time) overhead during compilation. Section 5.4 introduces additional, more advanced, code obfuscation techniques relevant for our library detection approach.

App Components Every application must define an app manifest called AndroidManifest.xml which provides essential information about the application to the Android system. It contains the package name that serves as a unique identifier and the set of requested permissions. In addition, it declares the set of app components that compose the application and defines the conditions in which they can be launched. In Android there exist four basic types of components:

- **Activities** are foreground tasks that implement user interface screens. They constitute the main entry points for users to interact with applications.
- **Services** perform long-running operations in the foreground (e.g. playing an audio track) or in the background (e.g. synchronizing data) and do not provide a user interface. They continue to run even if the user switches to different applications. Services may provide an IPC interface to which other services can bind and interact with it (see *bound services*).
- **Content Providers** manage access to data sources and provide granular control to access the stored data. Content Providers are the standard interface to connect data across processes. They provide a level of abstraction for accessing both structured data, e.g., an SQLite database, or unstructured data such as image files. Content Providers can be configured to restrict access to (subsets of) the data storage for different applications and can define permissions for reading and writing data.
- **Broadcast Receivers** are components that can be used to register for system and applications events. Once an event happens, all registered receivers for this event are notified by the Android system. For example, all applications that register for the event ACTION_BOOT_COMPLETED are notified once the system has completed the boot process.

Android apps adhere to a complex event-driven application lifecycle. Components can be asynchronously triggered by events or launched (and stopped) by user interaction. Each of these components maintains its individual lifecycle with predefined callback methods that are implicitly invoked by the runtime environment. Developers override these callback methods such as onCreate or onPause to initialize data structures, to save an app's state before closing it or switching it into the background. Figure 2.2 shows the complex Activity lifecycle with its predefined callbacks and state transitions.

Inter-Component Communication On Android, each app process is forked from a system process called *Zygote*. Zygote is initialized at system boot time and preloads

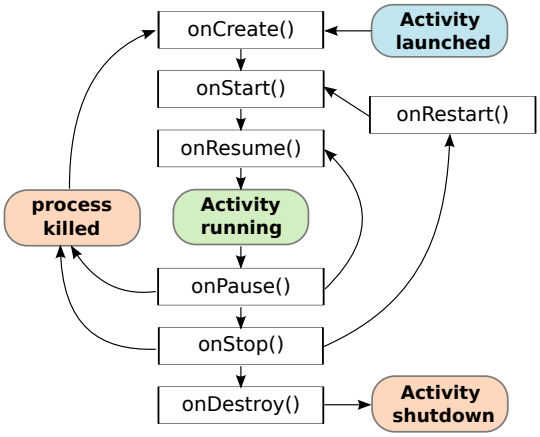


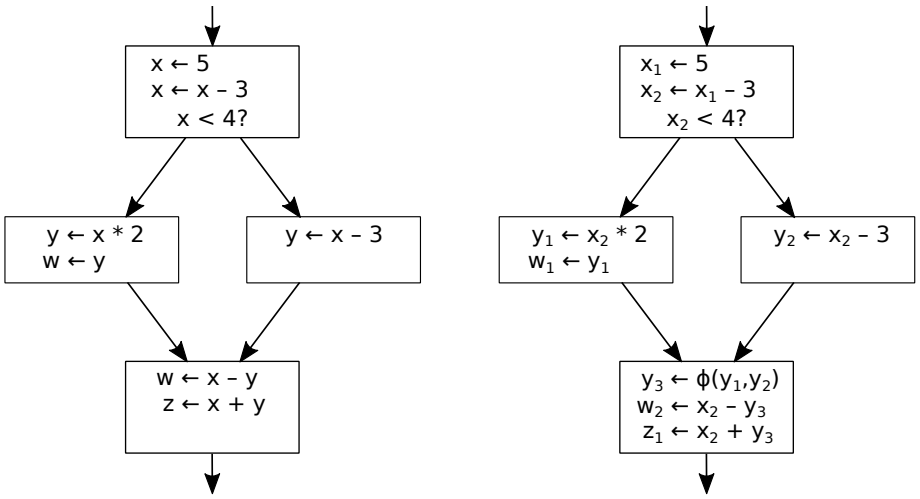
Figure 2.2: Activity lifecycle (adapted from (48)) with predefined callback methods.

common framework code and core system libraries. When an application is started, the app's code is loaded and run in new process forked from Zygote. All apps can access the same set of shared core libraries which consequently reduces the load time and results in faster application launch times. As a consequence, every app is executing in its own process with a distinct system identity, i.e. the Linux user ID (UID) and group ID (GID). Due to this privilege separation at operating system level, apps are, by default, isolated from each other and can not directly access data from another application. Since Android is based on Linux, processes can use traditional inter-process communication (IPC) mechanisms to communicate with each other, including the filesystem, sockets, or signals. The primary way to communicate with other apps, however, is the Android-specific IPC mechanism *Binder*—a lightweight and high-performance RPC implementation derived from OpenBinder[108]. Since Binder is used both for in-process and cross-process communication between app components, this is often referred to as inter-component communication (ICC).

To hide low-level implementation details of Binder-based ICC, Android provides multiple layers of abstraction. The most commonly used abstraction are *Intents*. Intents are simple message objects with two primary attributes—an *action* to be performed and the *data* to be operated on, typically given as a URI. The example ACTION_VIEW content://↷ contacts/people/ would display a list of people, which the user can browse through. The receivers are determined depending on the two types of Intents, explicit and implicit Intents. Explicit Intents address one specific receiver that needs to be defined via an additional attribute *component*. For unspecific implicit Intents, receivers are resolved by the Android system at runtime depending on the configured attributes.

2.2 Static Analysis Primer

This section introduces some common terminology and data models for static code analysis used throughout the following chapters.



(a) Control-flow graph

(b) CFG with instructions in SSA form

Figure 2.3: Control-flow graph with original instructions and instructions in SSA form.

Intermediate Representation Static code analysis approaches typically do not work directly on source code or binary-/bytecode, but first, translate it into an *intermediate representation* (IR) that abstracts from high-level syntactic sugar of programming languages and from low-level processor-specific instructions. In general, IRs are the language of an abstract machine used to represent code independent of a specific source or target language, e.g. the analysis frameworks WALA[77] and Soot[126] are both capable of transforming Java source code, Java bytecode, and dex bytecode into the same intermediate representation. Due to their abstraction and special properties, intermediate representations are especially suitable for code analysis and optimization tasks. One useful property is the *Static Single Assignment* (SSA) form[2, 116]. An intermediate representation is in SSA form, iff each variable is assigned exactly once and every variable is defined before being used. This is achieved by splitting existing variables into versions, i.e., the first definition of a variable x is renamed to x_1. Figure 2.3 shows a simple code snippet and its respective SSA form after variable renaming. Synthetic phi functions are added at the beginning of control flow merge points to resolve ambiguity. In Figure 2.3b the statement $y_3 \leftarrow \phi(y_1, y_2)$ indicates that the new definition y_3 can either take values y_1 or y_2 depending on the path taken. This versioning implies that use-def chains in the SSA form are explicit, i.e., their length is one. This makes the static single assignment form particularly appealing for code optimizations such as constant propagation or dead code elimination.

Data Models *Control-Flow Graphs* (CFG)[1] are an essential building block for many static analyses and compiler optimizations. In a CFG each node in the graph represents a basic block (BB), i.e. a straight-line set of instructions without jumps or jump targets. Directed edges represent jumps in the control-flow. CFGs capture the intra-procedural control-flow, i.e. the flow within one method. In Figure 2.3a the first BB ends with an

if-conditional and the control flow is split into a *then*-branch and an *else*-branch. Both branches subsequently merge again to a single execution path.

A *Call Graph* (CG) is an inter-procedural control-flow graph which represents caller-to-callee relationships between methods in a program. Each node represents a method and directed edges (f,g) indicate that a method f calls method g. Call graphs can be generated with varying levels of precision. A lightweight approach to generate a CG constitutes using a *Class Hierarchy Analysis* (CHA)[42], a type-based analysis to determine class inheritance. For Java, the class hierarchy is always rooted at the java.lang.Object class and allows to easily check for subclasses of a particular class, implementing classes for interfaces, or inherited methods. This is important for programming languages that feature dynamic dispatch where the class hierarchy can be consulted for potential receivers. If the set of possible receivers is larger than one, the call-graph over-approximates the runtime behavior. More precision can be added by conducting an alias/pointer analysis to (try to) statically determine the concrete runtime type at each call site. The increased precision typically comes at the drawback of longer generation times and larger storage requirements.

Besides control-flow, data and control dependencies between program statements are of special interest for static analysis. These can be modeled intra-procedurally using a *Program Dependence Graph* (PDG)[55]. A control dependence between statements s_1 and s_2 exists, if s_1 controls the execution of s_2 or vice versa, e.g. through if-conditionals or while-statements. Control dependencies are typically extracted from control-flow graphs. A data dependence between two statements exists if a definition of a variable v at one statement might reach the usage of v at another statement. *System Dependence Graphs* (SDG) are used to combine PDGs to model data and control dependencies inter-procedurally.

Analysis Techniques A *reachability analysis* is a code optimization approach to identify unreachable/dead code or loops by traversing edges of the control-flow graph or call graph. In code analysis, this technique is used to determine whether a code location, e.g. a sensitive sink statement, is control-flow reachable from an API or another statement, e.g. a sensitive source. A reachability analysis typically bootstraps a more expensive analysis when execution paths exist between points of interest. *Data and control dependence analyses* can similarly be conducted by traversing the respective edge types in a PDG/SDG.

Program slicing[145] is a technique to compute a set of statements, the *slice*, that may affect the values at some statement of interest, usually referred to as the *slicing criterion*. Slicing can be based on iterative data-flow analysis or on PDG/SDG graph traversal. A forward slice starts at a slicing criterion and computes all statements that may be influenced by this statement, while a backwards slice computes all statements that may influence the criterion. The generated slice represents a subset of the program that should still produce the same output for a given input. In contrast, a *taint analysis* traces the path of a labeled/tainted value through the application and observes all objects/values that are affected by the original value. Taint propagation rules define

how tainted values propagate through different types of instructions to enable tracking of labeled data during execution.

3

R-Droid

Leveraging Android App Analysis with Static Slice Optimization

3.1 Motivation

Today's feature-rich smartphone apps intensively rely on access to highly sensitive (personal) data. This puts the user's privacy at risk of being violated by overly curious apps or libraries (like advertisements). Central app markets conceptually represent the first line of defense against such invasions of the user's privacy, but unfortunately, we are still lacking full support for automatic analysis of apps' internal data flows and supporting analysts in statically assessing apps' behavior.

In this work, we present a novel slice-optimization approach to leverage static analysis of Android applications. Building on top of precise application lifecycle models, we employ a slicing-based analysis to generate data-dependent statements for arbitrary points of interest in an application. As a result of our optimization, the produced slices are, on average, 49% smaller than standard slices, thus facilitating code understanding and result validation by security analysts. Moreover, by re-targeting strings, our approach enables automatic assessments for a larger number of use-cases than prior work. We consolidate our improvements on statically analyzing Android apps into a tool called R-Droid and conducted a large-scale data-leak analysis on a set of 22,700 Android apps from Google Play. R-Droid managed to identify a significantly larger set of potential privacy-violating information flows than previous work, including 2,157 sensitive flows of password-flagged UI widgets in 256 distinct apps.

3.2 Problem Description

Modern smartphone apps offer an abundance of features that request access to the users' highly sensitive, personal data. The wide proliferation of these apps has made them a prime target for malware developers, and the variety of reported privacy incidents has fueled the legitimate privacy concerns of end users that their sensitive data is stealthily collected, monetized, and disseminated [65, 38, 56, 96]. Centralized app markets have responded by trying to identify malicious and overly curious applications even before these apps are deployed on a user's smartphone. To this end, they strive for comprehensive application vetting to understand app internals and to thereby identify abnormal app behaviors.

Static analysis of apps is widely accepted as a well-suited, automated concept for application security vetting on a large scale. In the context of Android, prior work has already successfully identified particular security and privacy problems like (user-intended) privacy leak detection [73, 59, 152, 151, 144], component hijacking vulnerability detection [94], and misuse of (framework) features such as the crypto API [52] or dynamic code loading [112], to name a few.

All these approaches share the common goal to precisely capture which data flows into security- and privacy-sensitive method calls. Unfortunately, existing static analysis approaches are (fully or partially) agnostic to the *concrete* data values that arise during execution of an app (i.e., strings or primitive values). This can make a crucial difference

in assessing an app as being harmless or dangerous to the users' privacy. For instance, the concrete value for a receiver number in a text message app tells apart a legitimate app from one that sends premium SMS messages. Some approaches have considered this problem of statically recovering concrete runtime values, but are currently limited to coarse-grained approximations for runtime strings [105, 73, 32, 35]. In more evolved cases, this typically results in conservative approximations such as (*"could be any string"* or *"any combination of these strings"*). As a consequence of this limitation, they face many *false positives* (i.e. false alarms) or are even not suitable for assessing such cases at all (see Section 3.4).

Another aspect that has received little attention so far is that any static analysis requires a significant amount of manual investigation either to validate the results or to understand *how* data is processed within the application code. However, manually investigating the output of existing approaches even for simpler cases—typically a huge list of data-dependent statements and an involved kind of formal security assessments—constitutes an intricate task, since existing tools either work with the app's bytecode [73, 112] or transform it into an intermediate representation [19, 144, 94, 59] that is usually less amenable to manual review than the original source code. As a consequence, the efforts for a human analyst in validating and assessing the results of current analysis solutions are significant.

3.3 Contribution

To address the aforementioned challenges, we present a novel slice optimization approach to facilitate privacy- and security assessments on Android applications. Our approach builds on a standard slicing-based analysis to generate data-dependent statements for arbitrary points of interest in an application. While existing app analysis approaches already assess apps based on these early results, our analysis proceeds by further optimizing the slice statically. Our optimization provides the following benefits: 1. A general purpose value analysis to precisely reconstruct values/strings to ease (semi-)automatic checks for more evolved security assessment tasks. 2. Our optimization pipeline statically transforms slices into semantically-equivalent, concise slices. This improves readability and reduces the number of false positives, a major benefit for an efficient reviewing process. The optimized slices can subsequently be further assessed by distinct *security modules* that create additional insights about sensitive data flows within the application and that better facilitate manual app reviewing. Technically, we make the following three contributions:

1. Novel Slice Optimization Approach. We start by leveraging a system dependency graph (SDG) that distinguishes different objects of the same type (object sensitivity), fields of the same object (field sensitivity), calling contexts of invoked methods (context sensitivity), and definitions of the same local variable on different paths through a method (flow sensitivity). We add a comprehensive application lifecycle model to faithfully take Android's peculiarities into account. On top of a slicing-based dependency analysis,

we propose a novel slice optimization approach that, on a high level, constitutes a use-def tracker to eliminate spurious dependencies that the data-dependency analysis failed to resolve and a comprehensive value analysis to re-assemble strings and primitive values that are passed as parameters to security-relevant functions. To this end, we adopt optimization techniques from partial evaluation, domain knowledge, and copy propagation to receive concise and semantically-equivalent slices with a low number of statements. As a result, we receive optimized slices that are about 50% smaller than the original slices, making any subsequent manual reviewing task more efficient. In addition, more evolved security problems such as premium number assessment (see Section 3.4) can be evaluated automatically due to our value analysis.

2. Complementary Analysis via Security Modules. Our approach supports the integration of further security modules to extend and amplify the analysis of apps' w.r.t. user's security and privacy. Security modules define their own data flow sources/sinks and receive the optimized slices to perform security assessments. In this work, we realize three such security modules—data leakage detection, user input propagation, and slice rendering for manual code review—that we used for our large-scale evaluation of Google Play apps.

3. R-Droid and Large-scale Evaluation on Google Play. We consolidate all aforementioned features into a tool called R-Droid. The evaluation of R-Droid on the widely accepted, open-source testsuite DroidBench excels over related work [19, 144, 63] with nearly optimal precision (97%, one false alarm) and 80% recall (7 missed violations). In a large-scale evaluation of 22,700 apps from Google Play, R-Droid managed to identify a significantly larger set of potential privacy-violating information flows than previous work, including 2,157 sensitive flows of password-flagged UI widgets in 256 distinct apps. Finally, we demonstrate the effectiveness of our approach to manual code reviewing on the common use-case of understanding malware behavior.

3.4 Technical Problem Description and Approach

We start with the illustrative example of *premium number classification*—a major monetization factor of malware—to demonstrate why current data leak detection and reachability analyses are insufficient for a faithful security analysis of Android apps. The example is depicted in Listing 3.1 as a code excerpt for sending premium SMS that many Android SMS malware variants rely on [161]. In line 10, an SMS with activation code *95pAHD* is sent (without user interaction) once the *MainActivity* is displayed. The premium receiver number is selected randomly and assembled via string concatenation, which constitutes a simplistic form of obfuscation.

Existing approaches based on static analysis do not resolve the concrete values for the receiver number of the text message (SMS), for different reasons. Forward analysis approaches [80, 151, 19, 59, 144, 63] rely on sensitive sources to execute their analysis,

Listing 3.1: Premium SMS example

```
1 public class MainActivity extends Activity {
2   protected void onCreate(Bundle savedInstanceState) {
3     String val1 = "10";
4     int val2 = 66953930;
5     if (Math.random() > 0.5d) {
6       val1 = "106618";
7       val2 = getValue();
8     }
9     SmsManager sm = SmsManager.getDefault();
10    sm.sendTextMessage(val1+val2, null, "95pAHD", null, null);
11  }
12  private int getValue() { return 5829; }
13 }
```

none of which are present in this example. Backward analyses approaches do not resolve the concrete numbers as well. Approaches tailored to data leak detection [19, 144] do not flag this example as critical, due to the absence of sensitive information, basic string analyses [73, 32, 105] are either limited to constants and/or lack path-sensitivity. Heuristic approaches simply retrieve string values from the bytecode and combine them in various ways to determine meaningful combinations. This will miss the implicit conversion of *val2*, an integer variable, to a string during concatenation (internally, this constitutes a StringBuilder.append call). Moreover, intra-procedural approaches [73] do not appropriately capture the number *5829*, which is returned by the method call getValue (it is a potential value as well since it is implicitly converted to a string). Moreover, even if all values can be identified, existing analyses do not determine the set of possible values per path. The common remedy is to enumerate all possible combinations of the identified strings, introducing false positives. In a nutshell, a path-sensitive value analysis is essential for determining the actual numbers *1066953930* and *1066185829*, which can, in practice, be compared against lists of known premium numbers.

3.4.1 Analysis Framework

The high-level architecture of our R-Droid is depicted in Figure 3.1. It comprises a pre-processing phase in which it collects application meta-data and generates a comprehensive application lifecycle model (cf. Section 3.4.1.2) that is subsequently used to create a system-dependence graph (SDG). For the analysis, we leverage a standard (backward) data-dependence slicer. It takes an arbitrary list of sinks (in terms of method signatures) as input to capture data that may influence these statements. The output is post-processed by our optimization pipeline to obtain semantically equivalent slices with a low number of statements and a lower number of false positives. These optimized slices can then be fed into customized security modules, e.g. for privacy leak or input propagation analysis. The implementation of R-Droid (blue colored boxes) comprises

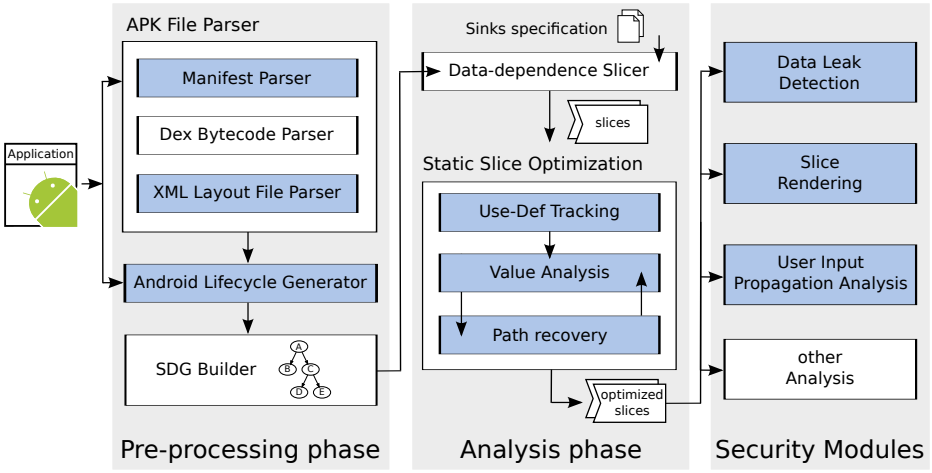


Figure 3.1: High-level architecture of R-DROID

approximately 11.5 kLOC, including code for the three security modules.

3.4.1.1 Pre-processing Phase

We start by extracting information included in the application package and analyze the manifest file that, among others, declares the apps' components and meta-data like the requested permissions, as well as layout files containing UI descriptions. To generate app data models, R-DROID leverages the information flow control (IFC) framework JOANA [69, 66]. Its frontend includes the analysis framework WALA [77], that offers an intermediate-representation (IR) in a static-single assignment (SSA) form [2, 116]. Their Dalvik frontend, adopted from SCanDroid [32], transforms Android bytecode directly into WALA's IR. We generate bytecode using dexlib [25] to model a static application lifecycle model for each app that takes the Android peculiarities into account.[1] The lifecycle-enhanced app bytecode is subsequently used by JOANA to generate an object-, context- and field-sensitive SDG. The SDG contains intra- and inter-procedural data- and control-dependencies (also for exceptions) as well as object-sensitive points-to information to resolve dynamic dispatch. To keep the complexity of the overall data structure tractable we do not include the full framework code. Instead, we use the lightweight framework model of the *DroidSafe* [63] project to capture data-dependencies within the framework.

3.4.1.2 Android Lifecycle Modeling

Android apps adhere to a complex event-driven application lifecycle that challenges static analysis approaches. Applications consist of multiple components that are asynchronously triggered by events, or launched (and stopped) by user interaction. Each of

[1] WALA's immutable IR precludes direct altering

Listing 3.2: Android lifecycle example

```
1  public class DataLeakage extends Activity {
2    private String deviceId;
3    protected void onCreate(Bundle savedInstanceState) {
4      new ATask().execute();
5    }
6    protected void onPause() {
7      deviceId = "fakeId";
8    }
9    // Button.onClick callback handler defined in XML file
10   public void leakId(View view) throws IOException {
11     File dir = Environment.getExternalStorageDirectory();
12     FileWriter writer = new FileWriter(dir);
13     writer.write("Device ID: " + deviceId);
14     writer.close();
15   }
16   private class ATask extends AsyncTask<Void,Void,String>{
17     protected String doInBackground(Void... params) {
18       TelephonyManager tm = (TelephonyManager)
19         getSystemService(Context.TELEPHONY_SERVICE);
20       return tm.getDeviceId();
21     }
22     protected void onPostExecute(String result) {
23       deviceId = result; }
24   }
25 }
```

these components maintains its individual lifecycle with predefined callback methods that are implicitly invoked by the runtime environment. Developers override these callback methods such as onCreate or onPause (cf. Listing 3.2) to initialize data structures, to save an app's state before closing it or switching it into the background. Moreover, event-listeners can be registered for services (e.g. location events are triggered whenever the device's location changes) or UI-events (e.g. the button-click handler leakId.)

To build an accurate lifecycle model, R-Droid starts by adopting the entry point discovery algorithm from [94]. Using the entry points discovered in the app's manifest file and the overridden component callbacks, R-Droid builds a callgraph and performs a code reachability analysis in order to identify new entry points—registered event-listeners and overridden framework methods. This process is repeated until convergence, i.e., until the entry point set reaches a fixed point. To improve precision over permitting lifecycle callbacks to be called in arbitrary order, R-Droid follows prior work [19, 144] and models individual component lifecycle methods that exploit the partial callback ordering. The resulting per-component models contain fewer invalid paths, which amends the precision of our subsequent analysis.

Finally, we generate a synthetic main method to connect the individual lifecycle methods. This method serves as single entry point for the analysis and mimics the initialization routine of an app that is executed on a real device. Concretely, the static class

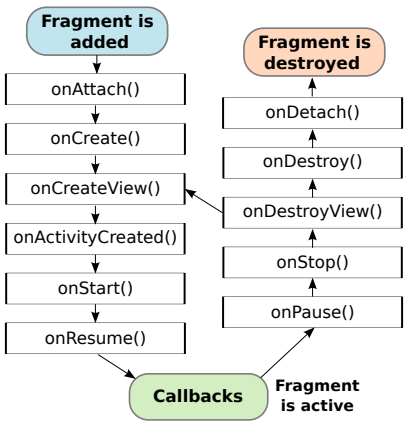


Figure 3.2: Fragment lifecycle

initializers are invoked first, with ContentProviders being the first components created during application launch [37]. After that, custom Application classes are invoked following the order provided by the class hierarchy. Finally, all other components can be executed in any order.

During our experiments, we found that the resulting model misses further lifecycle models for frequently used components: Fragments and Android's special threading class AsyncTask including parameter passing. These integral features either have not been considered by prior work or have been coarsely modeled, which results in false positives during analysis, i.e., invalid paths.

Fragment Lifecycle Android 3.0 introduced fragments as a design pattern to support more dynamic and flexible UI designs, which became imperative with the increasing number of tablets. Fragments can be classified as reusable sub-components of activities with a dedicated code base and an optional user interface. They maintain an individual lifecycle, receive input events, and can dynamically be added to and removed from a running activity via a FragmentManager object.

Fragments require a host Activity (FragmentActivity) for their execution. Similar to callbacks, fragments can either be statically added to the UI of an activity (via layout descriptions) or added/removed dynamically via FragmentTransactions. While parsing the layout files for Fragment declarations is straightforward, determining the concrete types in transactions requires points-to information. R-Droid generates this information along with the callgraph for the basic lifecycle model. As output, we receive a one-to-many mapping from Activity to Fragments. Event-handling for fragments is analogous to activities except for one case: Callbacks registered in a layout resource do not necessarily have to be declared in the fragment or in one of its inner classes, but can also be declared in its host activity. Identified callbacks can be discharged in arbitrary order while the fragment is running. For each identified Fragment R-Droid generates a lifecycle method in accordance with the official documentation [12] (see Figure 3.2) and subsequently adds it to the callback body of its host activity while active.

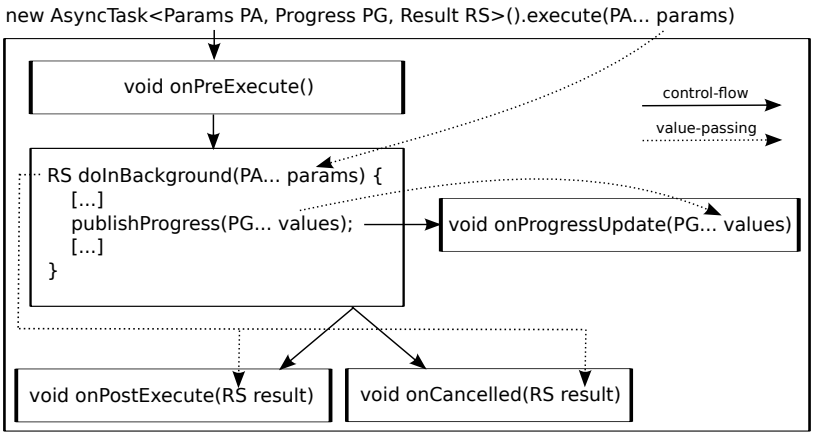


Figure 3.3: AsyncTask lifecycle

Modeling AsyncTask Android's application model imposes very strict response times on application components and forces developers to off-load potentially long-running code into separate threads. Apart from Java's default packages like java.util.concurrent, the Android SDK provides a dedicated thread class (AsyncTask) for outsourcing such code into background tasks (e.g., to download a file from the Internet) and feeding results back to the originating thread (e.g., as a progress monitor). Similar to Fragments, AsyncTasks are increasingly used by app developers (in our large-scale app evaluation we found 62.7% of 22,700 apps to include at least one AsyncTask). Hence a correct model of this feature is mandatory for static analyses to avoid false positives.

In contrast to Java's Thread class that contains a single entry method (run), AsyncTask features a series of callbacks that are discharged in a specific order once its execute command is invoked. To precisely model this behavior (including data passing between these callbacks) we propose the AsyncTask lifecycle depicted in Figure 3.3.

An AsyncTask is specified by three generic types, i.e. AsyncTask<Params,Progress,Result> that are used as argument and return types of its callback methods. If a task is triggered, the onPreExecute method is executed to set the task up. After that, the doInBackground method executes in a background thread to realize the main functionality. This method takes a *varargs* argument, basically corresponding to syntactic sugar for an array. Its return value is passed to the callback methods onPostExecute or onCancel, depending on whether or not the task was canceled. While the doInBackground method is executed, the method publishProgress can be invoked with a *varargs* argument that then triggers the callback onProgressUpdate with the same argument. R-DROID automatically identifies the concrete types for the generic parameter types (<Void,Void,String> in Listing 3.2) and generates a tailored lifecycle method to reflect this behavior. During analysis, these lifecycle methods are used instead of the framework's execute method.

Array arguments in callbacks constitute another challenge for static analysis. Section 3.4.2.3 explains in detail how R-DROID resolves them as part of the value analysis. This is the first work to generate a comprehensive model of concurrency pattern in

Android apps, including the complex and commonly used AsyncTask lifecycle with precise parameter passing.

3.4.2 Slice Optimization

Many related approaches [32, 80, 59, 151, 19, 144, 63] are essentially tailored to variants of privacy leak detection, i.e., detecting whether there is some flow/data-dependence between sensitive sources and sinks. Others focus on specialized forms of API usage [112, 106, 54, 52] which usually requires a precise reconstruction of values/strings for (semi-)automatic assessment. All these app vetting approaches underlie manual reviewing for result validation due to the absence of a ground truth. None of these tools provides dedicated support to facilitate such a reviewing process.

Hence, we propose a new slice optimization with the following benefits: 1. We provide a general purpose value analysis to precisely reconstruct values/strings to ease (semi-)automatic security checks. 2. Our optimization pipeline statically transforms slices into semantically-equivalent, concise slices. This improves readability and reduces the number of false positives, a major requirement for an efficient reviewing process.

Our slice optimization pipeline is designed as multi-stage post-processing technique. Given a standard data-dependence slice generated by JOANA, R-Droid first applies def-use tracking to eliminate spurious dependencies. Then, the value analysis is applied until convergence, i.e., no more optimizations can be applied. To allow for more aggressive optimization and value retargeting, R-Droid tries to reassemble execution paths within the slice to re-apply the optimizations on each path on success.

3.4.2.1 Instruction Filtering

To precisely reason about data-flows through the Android framework, it is imperative to include (parts of) the Android SDK/framework code during analysis. This implies that slices include instructions from within the Android API. Since our definition of sensitive sources and sinks is based on Android API signatures, this internal code distracts a human investigator and does unnecessarily bloat the resulting slices. Therefore, we first filter any instruction that does not originate from application code. This can be trivially achieved by loading application and framework code via different classloaders and subsequently filter instructions by classloader. This follows the idea of slicing with barriers [82] in which the slicer stops when pre-defined conditions are met, e.g., leaving application code.

3.4.2.2 Use-def Tracking

Traditional SDG-based slicing approaches [74] result in flow-*in*sensitive slices. However, as illustrated in Listing 3.1, flow and path information is essential to recreate exact values for automatic security assessments. Moreover, the resulting slices might contain

spurious dependencies, i.e., statements that do not directly influence arguments of the sink. This is because the slicer resolves data-dependencies for complete statements (such as method calls) and not for subsets of arguments. This usually leads to large slices and may additionally introduce false-positives.

We, therefore, conduct, as first optimization step, a use-def analysis on the resulting slices. Starting from the sink statement we iteratively backtrack register and field references and add defining and dependent statements to the result. To this end, we leverage WALA's SSA-based IR, in which use-def chains are explicit, i.e., for each use, there is exactly one definition. We add flow-sensitivity by ordering the statements in reverse control-flow order. While backtracking explicitly adds flow-sensitivity for defining statements, statements such as multiple calls on the same object can be ordered using flow information obtained by control-flow graphs (CFGs) of the enclosing methods (cf., list operations in Output 3.3). The resulting slices are more concise and readable since non-relevant statements have been eliminated.

Output 3.1: Reverse control-flow-ordered slice of DataLeakage example after use-def tracking

```
1  java.io.FileWriter→write{v6}(v16)
2  java.io.FileWriter→<init>{v6}(v4)
3  v6 = new java.io.FileWriter
4  v4 = android.os.Environment→getExternalStorageDirectory()
5  v16 = java.lang.StringBuilder→toString{v13}()
6  v13 = java.lang.StringBuilder→append{v8}(v11)
7  java.lang.StringBuilder→<init>{v8}("Device ID: ")
8  v8 = new java.lang.StringBuilder
9  v11 = DataLeakage{this}.deviceId
10 DataLeakage{this}.deviceId = "fakeId"
11 DataLeakage{this}.deviceId = p1
12 Entry DataLeakage$ATask.onPostExecute(java.lang.String)V
13 DataLeakage$ATask.onPostExecute{this}(v3)
14 v3 = DataLeakage$ATask.doInBackground{this}()
15 return v12
16 v12 = android.telephony.TelephonyManager→getDeviceId{v8}()
17 v8 = DataLeakage→getSystemService{v2}("phone")
18 v2 = DataLeakage$ATask{this}.this$0
```

Output 3.1 shows the slice of the *DataLeakage* example in Listing 3.2 after use-def tracking. Beginning at the sink statement, i.e., the write method (line 1), data-dependent statements are iteratively added in reverse-control flow order. An Entry meta statement (line 12) is added since the second field assignment (line 11) depends on the first argument p1 of the onPostExecute method. The string written to the SD card is assembled via a series of invocations on a StringBuilder object (lines 5-8). Its value depends on the field *deviceId*, that can have two values depending on the activity state: either the constant string *fakeId* (line 10) or the actual device id (line 16) is accessed in the instance of the AsyncTask class *ATask*.

3.4.2.3 Value Analysis

Our value analysis (generalized string analysis) statically simplifies complex expressions and re-assembles strings beyond constant values. It currently comprises four optimization steps that leverage techniques from partial evaluation, domain knowledge, and copy propagation. Execution path recovery is performed whenever possible to allow more aggressive optimization. The optimization pipeline is iteratively applied until convergence, i.e. until no more slice statements are modified. The outcome are semantically equivalent slices that contain fewer but more expressive instructions (due to retargeted strings and values). These optimized slices ease manual reviewing and allow a larger range of security assessments to be performed (semi-)automatically.

Copy Propagation. We adopt WALA's copy propagation to eliminate assignment statements (that may occur as result of the other optimizations) and copy constants/registers directly to the statements in which they are used. This also applies to values stored in and later retrieved from class fields. Moreover, R-Droid eliminates function calls that return constant values/references, such as getter methods.

Evaluating Unary/Binary Operations. This step statically evaluates unary and binary operations. Further optimizations like resolving indices for arrays and lists depend on this optimization. R-Droid uses points-to information to determine the type of an operation such as `int` or `double`. We then statically evaluate such operations iff the operand values are constants or can be iteratively resolved, e.g., an integer addition $x = 17 + 23$ is evaluated to $x = 40$. Static evaluation fails if at least one operand is non-constant, e.g., the result of a framework call to `Math.round()`. In this case, we cannot simplify the slice without expert knowledge.

Array Access Resolution. Related work on Android [73, 59, 151, 19, 144, 63] cannot precisely resolve array indices and thus over-approximates array modifications. This does not only reduce precision but also results in false alarms when sensitive data is written at position `x` but is later leaked from position `y`. Albeit challenging, resolving array access statically significantly improves the precision of the analysis. This particularly applies to the `AsyncTask` array parameter as described in Section 3.4.1.2. R-Droid resolves array accesses as follows: Based on its control-flow ordered list of array update instructions for every execution path, it statically reconstructs the content for each access and resolves the respective index. Three outcomes are possible:

1. If the access index i is statically computable and the data can be unambiguously determined at position i, then R-Droid can precisely determine the accessed content. It discards the array instructions and replaces them with the value at position i.
2. If the access index is statically computable for a position that can hold different data at the time of the access, R-Droid returns a list of possible values.[2]

[2]For instance, if the array is both updated with statically computable and non-computable indexes.

3. If the index is not statically computable, R-DROID returns a template defining how the index is computed and a string representation of the reconstructed array. In case that domain knowledge or a human expert *cannot* resolve this access further, all possible values must conservatively be considered in a subsequent security analysis.

Output 3.2 shows the relevant part of the slice for the testcase ArrayAccess2 of the benchmark suite DroidBench (see Section 3.6.4). This testcase evaluates if an analysis over-approximates array operations. An array is filled with sensitive (device Id, line 3-4) and non-sensitive data (*"no taint"*, line 2), of which the latter is finally leaked via SMS. Conservative algorithms will spuriously report a sensitive data leak. In contrast, R-DROID can statically resolve the array access. Given the array register v8 and the instructions that update the array (line 2-3), the content is statically reconstructed as follows:

Step	Instruction	Reconstructed Array
0	∅	[ ]
1	v8["4"] = "no taint"	[x,x,x,x,"no taint"]
2	v8["5"] = v16	[x,x,x,x,"no taint",v16]

The index v25 in line 1 is computed by a series of operations in the calculateIndex method (line 6-10). The expression evaluator starts with the return statement and iteratively assembles and solves the expression $(((1 + 1) * 5)\%10) + 4 = 4$. Thus, the non-sensitive value *"no taint"* is assigned to v8["4"] in the reconstructed array. With this semantic-preserving assessment, we correctly classify this test as non-leaking.

Output 3.2: Partial slice of the ArrayAccess2 DroidBench testcase

```
1  v27 = v8[v25]
2  v8["4"] = "no taint"
3  v8["5"] = v16
4  v16 = android.telephony.TelephonyManager→getDeviceId{v3}()
5  v25 = de.ecspride.ArrayAccess2→calculateIndex{this}()
6  return v10
7  v10 = v8 + "4 l"
8  v8 = v6 % "10 l"
9  v6 = v4 * "5 l"
10 v4 = "1" + "1 l"
```

Domain Knowledge. Retargeting values and strings in presence of Android API methods requires an understanding of the API semantics. For example, the concrete string value of the SMS receiver number (*"1066953930"* in Listing 3.1) is internally constructed via calls to the StringBuilder constructor and the append method. To enable automatic reasoning for such API methods we manually model their semantics as domain knowledge, e.g., in this case, the final string is the concatenation of the provided arguments. This knowledge is represented as rules specifying method signatures and

semantic descriptions how method arguments are transformed. These rules are then applied to (control-flow ordered) sequences of API calls on the same object (call chains).

Besides modeling StringBuilder and StringBuffer operations we further add domain knowledge for commonly used Java collection classes including various kinds of List, Map, and Set implementations. Internally, they behave like arrays and provide convenience functions for the developer. We encode the getter/setter methods of these classes as domain knowledge and handle them analogously to arrays. Output 3.3 shows an excerpt of a slice that creates a list object, subsequently adds non-sensitive (abc) and sensitive data (v20) and finally retrieves the first element. Without domain knowledge, the slice would be incorrectly flagged as a sensitive data leak. Adding semantic rules for the add, get and init methods, we can reconstruct the list content and output the correct string *"abc"* from the first position. This reduces the number of false positives when sensitive and non-sensitive data is stored in the same collection. If the argument of the getter method is statically not computable, or if there is no semantic rule for a framework method in the slice, we conservatively return the original instructions.

Output 3.3: Slice containing list operations

```
1 v34 = java.util.LinkedList→get{v7}("0")
2 v27 = java.util.LinkedList→add{v7}("def")
3 v23 = java.util.LinkedList→add{v7}(v20)
4 v12 = java.util.LinkedList→add{v7}("abc")
5 java.util.LinkedList→<init>{v7}()
6 v7 = new java.util.LinkedList
7 v20 = android.telephony.TelephonyManager→getDeviceId{v16}()
8 v16 = MyActivity→getSystemService{this}("phone")
```

In total, we modelled 104 methods in 23 classes as domain knowledge. This allows retargeting more complex values/strings from low-level API calls and thus increases the number of security checks that can be performed automatically. Moreover, we further simplify the slice and increase the precision of the result in the aforementioned cases.

3.4.2.4 Path Recovery

Being able to discriminate execution paths within the slice allows for a more aggressive optimization since register value ambiguity (e.g. due to branching) might be resolved. R-Droid detects three branching indicators within the slice: phi-instructions, callee-to-caller information and field value access/update relationships. WALA's IR offers flow-sensitivity for local variables due to the SSA form. Phi statements, an essential building block of an SSA form, are located at intra-procedural control-flow merge points and allow tracking of variable modifications across different paths. Field setter/getter operations are not tracked by the intra-procedural phi instructions and have to be handled separately. For the latter two branching indicators, multiple execution paths exist if there is a one-to-many mapping, i.e. if there is one field retrieval and multiple associated field update instructions or a non-API function is called from multiple

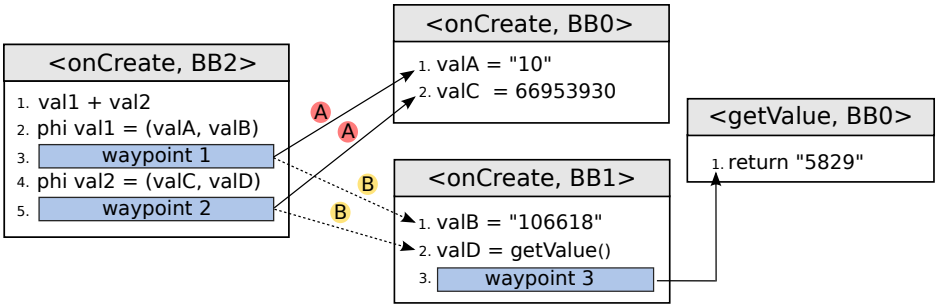


Figure 3.4: Path tree for receiver number in Listing 3.1

locations. Depending on the complexity of the slice this step might introduce a large number of execution paths or might not be feasible at all. To maintain a good cost-to-benefit ratio, a threshold for the maximum number of paths and processing time is configurable. If one of these values is exceeded the optimization is stopped, otherwise, the aforementioned optimization steps are re-applied to each execution path slice.

If R-Droid detects path information, it transforms the slice into a tree presentation (*path tree*). If this fails, e.g. due to recursive code within the slice, we terminate this optimization step. Tree nodes constitute basic blocks (BB)—code fragments with only one entry and one exit—from CFGs of the instructions' enclosing methods. Slice statements are subsequently mapped to their BB. Figure 3.4 shows the path tree generated for the receiver number in Listing 3.1 (two execution paths) with four nodes. Our tree model differs from CFGs in that instructions are connected rather than blocks, since the smallest unit of resolution are instruction arguments. If consecutive instructions of the slice reside in different basic blocks, we generate a waypoint. For phi-instructions at control-flow merge points, the waypoint has outgoing edges for each successor, e.g., waypoint1 points to the assignment instructions of valA and valB.

During tree traversal, the path extraction algorithm traverses the same outgoing edges on waypoints of the same node (either A or B). This prevents the generation of invalid paths through impossible combinations, e.g., in our example only two out of four distinct paths are feasible. Outgoing edges of waypoints are traversed before its immediate successor within the same node (if any). Within a node, the algorithm stops if the end of the instruction list or an instruction that is pointed to from a different node is reached. Applying this algorithm to Listing 3.1 yields two paths (which in turn are flow-sensitive slices) that contain the instructions to reconstruct the correct receivers.

3.5 Security Modules

R-Droid is a generic analysis framework tailored to the specifics of Android that provides optimized data dependence traces for arbitrary sinks in an application. Custom analysis modules leverage these traces to implement different security and privacy analyses. We demonstrate its effectiveness and relevance by defining three such security

modules—data leakage detection, user input propagation, and slice rendering. While the first two check the slices for sensitive sources, i.e. APIs to access sensitive data, the latter one eases the task of manual reviewing by transforming the statements into a format that is easier to read.

Further uses-cases (not implemented in this work) include API usage analyses for sensitive operations such as crypto and SSL APIs and checks for premium SMS numbers (cf. Listing 3.1). Such modules leverage the re-assembled runtime values—Strings and primitive values—generated by the value analysis to automatically check whether passed arguments follow security and privacy best practices.

3.5.1 Data Leakage Detection

Declared app permissions only indicate what an app could potentially do, but do *not* adequately capture the *actual* behavior of the app. In particular, correlations between permissions are hidden, e.g., if address book data leaks to the Internet. Consequently, users cannot appropriately assess the risk entailed by installing an app [113, 20, 64]. Our data leakage detection module reports source-sink pairs identified between permission-clad APIs.

We leverage the sources and sinks of *SuSi* [18], and extend it as follows: Sinks from the Apache classes HTTPClient and their subclasses are added as they often constitute substantial parts of the app's network communication. We add sources that are (a) argument-dependent or (b) constructed via a series of method calls. A prominent example for (a) is the API to access the secure system settings, in which the sensitivity of the returned data depends on the argument. The device ID is frequently accessed (via argument "android_id") and serves as a unique identifier. Our module classifies the sensitivity of the result according to the parameter resolved during value analysis. The locale device settings are an example for (b), as getLanguage() and getCountry() only represent sensitive information when accessed via java.util.Locale->getDefault() (that typically holds the user's preferred locale). Finally, we include framework fields that provide sensitive data into our list of sensitive sources. android.os.Build and its subclasses provide numerous data about the installed Android build and version. To preclude false positives, we disregard sinks for which the app lacks the required permission (libraries frequently probe for permissions of the host app) using the permission maps generated by *axplorer* (see Chapter 4).

3.5.2 User Input Propagation Analysis

User input propagation is a specialization of the previous module that focuses on leaks of user data provided via UI widgets. It is configured with the same sinks, however, we identify sources from UI input accessed via findViewById of android.app.Activity, android.view.View, and their subclasses. Input fields marked as passwords are of particular interest, hence we check for the respective view attributes. Slices that include user input may be forwarded to the slice-rendering module to improve readability and to facilitate

manual assessment. Recent work [76, 100] presented an approach to cover a wider range of sensitive user inputs (like credit card inputs) by inferring input widget sensitivity via UI layout descriptions including labels and hints. Integrating such approaches is an interesting future extension.

3.5.3 Slice Rendering Module

Related work is usually limited to answering *whether* data may flow but has limited support for answering *how* it flows since they do not account for individual execution paths between a source and a sink. Our evaluation shows that this is insufficient as traditional slices usually contain a significant fraction of the original program (we found on average 33 statements/slice and examples with up to 4k statements), which impedes manual inspection.

Our optimization pipeline efficiently reduces the number of instructions per slice. Still, reading bytecode or instructions in some intermediate representation is not as convenient as reading source code. To minimize this gap, this module renders the optimized slices in a human-readable format. To this end, we transform statements into series of call chains on the same object. Concretely, the algorithm starts at the sink and collects all invocations on the target object, reorders them according to runtime execution order and iteratively merges them. Class names are omitted when the class is constant for consecutive invocations. To structure the output, we add assignment statements to store call chains in variables, when they are subsequently used as arguments of framework API calls. The result omits redundant information and improves readability (see Section 3.6.4).

3.6 Evaluation

We evaluated R-Droid on the original DroidBench testsuite and received nearly optimal results. To demonstrate the scalability and utility of our approach we conducted a large-scale data/input leakage analysis on 22,700 apps from Google Play in which R-Droid identified a large number of privacy-critical data flows originating from sensitive APIs or from UI widgets. Finally, we elaborate on our slice optimization and the effects of our slice rendering module on manual reviewing efforts.

3.6.1 DroidBench Test Suite

DroidBench is an evolving open-source test suite [50] containing Android apps crafted to evaluate static and dynamic analysis approaches. Many of these synthetic test cases evaluate the recall of the performed analysis, hence there is a bias towards over-approximating approaches. There is only a small number of cases that explicitly check the precision, i.e., whether the analysis reports on the actual flows and does not generate false positives. We test R-Droid on the widely used original DroidBench (v1.0) to

demonstrate the effectiveness of our lifecycle modeling and the benefits our our slice optimization on the few cases the pose such a challenge. We compare our results with FlowDroid [19], AmanDroid [144], and DroidSafe [63]. For the sake of completeness, we also report the results for the four implicit-flow testcases that are missing in the original paper [19]. Table 3.1 shows the detailed results for all testcases.

FlowDroid achieves a high precision (87%, 4FPs) and reasonable recall (74%, 9 missed flows). Amandroid, that adapts their lifecycle modeling, offers similar precision and recall as FlowDroid without being explicit about the failing testcases. DroidSafe uses a novel framework abstraction to capture data-dependencies within the API. As a result, they achieve the highest accuracy among the analyses tools (83%, 6 missed flows). However, their approach also erroneously reports four sensitive flows due to their flow-*in*sensitivity and missing lifecycle modeling. Instead, our approach yields a nearly optimal precision of 97% (1FP due to a missing inter-procedural must-alias analysis) and 80% recall. R-Droid particularly excels in the category *ArraysAndLists* that tests over-approximation for container classes, where all other tools fail. Instead, our value analysis can successfully track data written to and accessed from such containers. Hence, R-Droid classifies them correctly as non-leaking.

3.6.2 Data Leakage Analysis

Even though benchmarks like DroidBench are a valuable tool to compare different approaches, they are hand-crafted and their coverage of functionality is rather limited, given the modest number of tests. Therefore, we conducted a large-scale data leakage analysis on 22,700 apps from the Google Play Store. The apps were crawled between August 20-23, 2014, starting from the top 100 of each category and iteratively crawling recommended and similar apps. For this evaluation, we configured R-Droid's path recovery with a 3 min timeout and a maximum number of 32 paths. During analysis, this affected 19% of all sinks. In these failing cases, the slices were assessed after the first value analysis run, which might cause imprecision. The experiments were conducted on a server with four Intel Xeon CPU E5-4650L @ 2.60GHz processors with 8 cores each and 768GB RAM on which we ran 64 *single-threaded* analyses in parallel. Despite its precision, R-Droid had a reasonable average processing time of 26 minutes of which the graph builders consume about 90%.

Our experiments emphasize the relevance of fragments in current apps (33% contain at least one). As an increasing fraction of code has been moved from activities to fragments, modeling their lifecycles accurately becomes imperative for precise static analyses. Our evaluation shows that apps, that were originally published prior to Android 3.0, often still do not adhere to the recommended fragment-based layout design. For all other apps, however, we detected 11 fragments on average, which corresponds to the average number of activities per app. Similarly, AsyncTasks are becoming the standard for network communication. 14,244 apps (62.7%) include at least one AsyncTask.

Table 3.2 summarizes our findings as flows from sensitives sources to sinks (grouped by categories as provided by the SuSi project). We report the absolute number of

⊛ = correct warning, ⋆ = false warning, ○ = missed warning
multiple circles in one row: multiple leaks expected
all-empty row: no leaks expected, none reported

App Name	Fortify	FlowDroid	AmanDroid	DroidSafe	R-Droid
Arrays and Lists					
ArrayAccess1		⋆	⋆	⋆	
ArrayAccess2	⋆	⋆	⋆	⋆	
ListAccess1	⋆	⋆	⋆	⋆	
Callbacks					
AnonymousClass1	⊛	⊛	⊛	⊛	⊛
Button1	⊛	⊛	⊛	⊛	⊛
Button2	⊛ ○ ○	⊛ ⊛ ⊛ ⋆	⊛ ⊛ ⊛	⊛ ⊛ ⊛	⊛ ⊛ ⊛ ⋆
LocationLeak1	○ ○	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛
LocationLeak2	○ ○	⊛ ⊛	⊛ ⊛	⊛ ⊛	⊛ ⊛
MethodOverride1	⊛	⊛	⊛ ⋆	⊛	⊛
Field and Object Sensitivity					
FieldSensitivity1					
FieldSensitivity2					
FieldSensitivity3	⊛	⊛	⊛	⊛	⊛
FieldSensitivity4					
InheritedObjects1	⊛	⊛	⊛	⊛	⊛
ObjectSensitivity1					
ObjectSensitivity2				⋆	
Inter-App Communication					
IntentSink1	⊛	○	⊛	⊛	⊛
IntentSink2	⊛	⊛	⊛	⊛	⊛
ActivityComm1	⊛	⊛	○	⊛	⊛
Lifecycle					
BroadcastRecvLifecycle1	⊛	⊛	⊛	⊛	⊛
ActivityLifecycle1	⊛	⊛	⊛	⊛	⊛
ActivityLifecycle2	⊛	⊛	⊛	⊛	⊛
ActivityLifecycle3	○	⊛	⊛	⊛	⊛
ActivityLifecycle4	⊛	⊛	⊛	⊛	⊛
ServiceLifecycle1	○	⊛	⊛	⊛	⊛
General Java					
Loop1	○	⊛	⊛	⊛	⊛
Loop2	○	⊛	⊛	⊛	⊛
SourceCodeSpecific1	⊛	⊛	⊛	⊛	⊛
StaticInitialization1	⊛	○	⊛	⊛	⊛
UnreachableCode1	⋆				
Miscellaneous Android-Specific					
PrivateDataLeak1	○	⊛	⊛	⊛	⊛
PrivateDataLeak2	⊛	⊛	⊛	⊛	⊛
DirectLeak1	⊛	⊛	⊛	⊛	⊛
InactiveActivity	⋆				
LogNoLeak					
Implicit Flows					
ImplicitFlow1	○	○	○	⊛	○
ImplicitFlow2	○ ○	○ ○	○ ○	○ ○	○ ○
ImplicitFlow3	○ ○	○ ○	○ ○	○ ○	○ ○
ImplicitFlow4	○ ○	○ ○	○ ○	○ ○	○ ○
Sum, Precision, and Recall					
⊛, higher is better	17	26	27	29	28
⋆, lower is better	4	4	4	4	1
○, lower is better	18	9	8	6	7
Precision $p = ⊛/(⊛ + ⋆)$	81%	87%	87%	88%	97%
Recall $r = ⊛/(⊛ + ○)$	49%	74%	77%	83%	80%
F-measure $2pr / (p + r)$	0.61	0.80	0.82	0.85	0.88

Table 3.1: DROIDBENCH v1.0 test results (as reported in (19, 144, 63))

Sinks by category	Sensitive sources by category														
	Account Info	**Build Info**	**Bluetooth Info**	**Calendar**	**Contacts**	**Database**	**File Info**	**Locale Info**	**Location**	**Network Info**	**NFC info**	**SMS-MMS**	**Unclassified**	**Unique Identifier**	**Version Info**
Code Loading	12	477	27	153	–	388	17	328	184	471	1	1	91	376	205
File	2	16	2	23	–	36	–	7	9	37	2	2	10	19	3
Log	146	3,563	285	1,516	2	3,648	83	2,501	1,676	4,401	31	29	1,031	3,148	1,570
ICC	1	10	2	34	–	42	–	11	5	110	–	1	7	12	7
Network	49	1,017	121	505	–	1,099	22	803	493	1,255	6	12	302	897	478
SMS-MMS	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Unclassified	202	6,335	389	2,305	13	5,034	127	4,797	2,715	6,137	49	38	1,388	4,618	2,921

Table 3.2: Absolute numbers of apps with data flows from sensitive sources to sinks grouped by category from our set of 22,700 apps.

apps with flows between source and sink category from our set of all analyzed apps. Our findings confirm the common belief that highly sensitive data like user location or unique IDs are frequently accessed and leaked via various channels. As an example, unique IDs are frequently written to logs (13.8% of all apps) and to the network (3.9%). There is a significant number of flows to unclassified sinks. Their classification depends on the runtime type of the receiver and requires a dedicated analysis. For example, the write methods of an OutputStream may write data to memory, to a file or to the network depending on the concrete stream instance. Similarly, contact information can be accessed in multiple ways using adapters or ContentResolvers that are queried with a specific data URI. Resolving this URI in a pre-processing step is required to classify calls to these classes correctly, e.g., as a sensitive source. Locale, version, and build information have been widely ignored as a sensitive source. Besides common cases in which such information is written to system logs for debugging purposes, manual investigation revealed that a combination of this information is commonly used to generate a unique user-agent ID to identify and track the user. This clearly indicates that access to such low-sensitive data (that is not protected by permissions) may threaten the user's privacy when combined to create a unique identifier. This emphasizes that systems that perform privacy leak detection solely based on API calls are likely to miss such cases since version and build information are stored in static fields. R-Droid did not find sensitive data flows to SMS–MMS sinks. Reasons are the small number of apps that declare the required permissions (2.6% in our dataset) and that SMS is not suitable to transmit large amounts of data. In fact, most of these apps can be classified as phone finder/tracker that notify the owner via SMS about odd incidents. Although ICC modeling is out of scope for this work, we quantified how often sensitive data reaches API methods that initiate ICC, such as startActivity. The absolute numbers are relatively small. The fraction of cases that actually leak the received data is presumably even smaller, implying that complex ICC-related privacy leaks rarely appear in applications.

		Sources		
		Location	**Network Info**	**Unique Identifier**
Sinks	**Log**	16.1%	26.5%	42.1%
	Network	4.8%	7.6%	12.5%
	Unclassified	26.1%	0.8%	61.7%

Table 3.3: From all apps that hold the required permissions to call source and sink APIs, this table shows the percentage of apps that have at least one data flow from source to sink, i.e., from all apps that require the INTERNET and location permissions, 4.8% leak the location data via the Internet.

Using publicly available API-to-permission information [20, 18], we compute the percentage of apps that have a sensitive data flow between sources and sinks that require a permission with respect to all applications in the dataset holding these permissions. Table 3.3 shows that more than one quarter of all apps requiring the android.permission.↷ ACCESS_NETWORK_STATE permission, leak network information via the system logs. 12.5% of apps holding both permissions android.permission.READ_PHONE_STATE and android.↷ permission.INTERNET indeed leak a unique identifier to the Internet. The high number of flows to *Unclassified* sinks again emphasizes the need for an improved classification algorithm that is executed prior to the analysis to triage relevant sinks only.

3.6.3 Leakage of Sensitive User Input

Private data may not necessarily originate from API calls, but may also be inserted by the user via user interface widgets like TextInputs. Detecting such data flows requires a dedicated analysis as described in Section 3.5.2. Applied to the Google Play test set R-DROID reported a total number of 2,157 flows of password-flagged UI widgets in 256 distinct apps. We manually validated the flows in 150 apps. 9% of all flows were erroneously flagged as leaking. Some of these false positives could be eliminated by deriving more export knowledge. Although not confirmed by a user study, the effort necessary to manually investigate these cases was notably lower due to the smaller and more concise output. Similarly, sorting instructions by control-flow helps to read and understand the output (see Section 3.6.4). We exemplify our findings in the following:

Case Study: Logging of Private User Data. We identified that developers frequently log data that is supposed to be written to files or the network. Although this facilitates debugging during development, it may be considered a privacy breach by users if sensitive input is included in the log. The app com.bitsontherun.android.dashboard uploads videos to the *Bits on the Run* online video platform. Among others, the server response of a sign-up attempt, including password and email address from an UI widget in plain, is written to the system logs.

Case Study: Plain User Input Transmission via Insecure Channels. User-provided input is frequently sent to remote servers via HTTP. As part of its service registration com.CG.checkgmov2 sends a POST message including first and last name, email address, and password in plaintext. This data originates from UI widgets, where one is marked as password. com.camilo.hkingorders manages/tracks purchases made on *hobbyking.com*. Within the login routine, R-DROID reconstructed the authentication request as follows: http://www.hobbyking.com/hobbyking/store/uh_customerAuthenticateExec.asp?email=$EMAIL&password=$PW. Both placeholders denote unsanitized user-provided text inputs accessed via Activity→findViewById($resourceId). This clearly violates the user's privacy, since we manually verified that the webshop supports HTTPS. We also found several cases in which user data is transmitted via the Facebook graph library. com.bokskya.books integrates this API to publish user feeds. The widget used to enter status messages is marked with the password flag by the developer since such messages could include private information. However, R-DROID reported that these messages are sent to Facebook via HTTP. Manual investigation of the API documentation revealed that the library indeed does not support secure transmission via HTTPS (at that time). Unfortunately, the app does not warn the user to not enter sensitive information.

3.6.4 Support for Manual Investigation

As shown in this section, additional manual investigation is often required to either validate findings or to understand app behavior in detail. R-DROID's optimization pipeline supports such efforts in multiple ways. In our Google Play evaluation, the slice optimization generated semantically-equivalent slices that are 49% smaller than standard slices on average. The initial use-def tracking accounts for 34% fewer instructions per slice on average. The multi-step value analysis further reduces the output size by almost 25%. This is due to the fact that most optimizations like string assembly, binary operation calculation, or array access resolution involve multiple instructions that are, in the best case, optimized to a single value. In many cases, the final slice contains less than 15 instructions. In addition, the control-flow ordering of instructions and the generic string/value assembly facilitate code understanding.

In the following, we elaborate on our slice rendering module, which transforms the optimized slices into readable and structured output to further ease manual investigation. To deduce malware functionality it is commonly necessary to manually analyze and understand its code. Our example (from the Malware Genome Project [161]) belongs to the jSMSHider family. This SMS malware targets Android users with a custom ROM. As these devices are already rooted the malware can install packages without the user's explicit consent. Output 3.4 shows the slice for a "command execution" sink after use-def tracking. (for the sake of brevity we replaced the original package name by *hider*). Although the partially-optimized slice is moderate in size, it shows that with an increasing number of instructions manual analysis becomes tedious. The slice contains a total of three execution paths. The argument p2 (line 1) of the sink statement depends on the method argument of the execCommand1 (line 3) which has two caller sites (line 4+11). The invocation on line 11 depends on the method argument of

runRootCommand1 in line 19 which again has two caller sites (line 20+21). The path recovery module (see Section 3.4.2.4) uses this information to split the list of instructions into slices per execution path as shown in Output 3.5.

Output 3.4: CF-ordered slice of package installation after use-def tracking

```
1  v10 = java.lang.Runtime→exec{v7}(p2)
2  v7 = java.lang.Runtime→getRuntime()
3  Entry hider.InstallService.execCommand1 (android.content.Context,java
       .lang.String)Z
4  v27 = hider.InstallService→execCommand1(p1, v24)
5  v24 = java.lang.StringBuilder→toString{v21}()
6  v21 = java.lang.StringBuilder→append{v17}(p2)
7  java.lang.StringBuilder→<init>{v17}("pm uninstall ")
8  v17 = new java.lang.StringBuilder
9  Entry hider.InstallService.uninstallapp (android.content.Context,java
       .lang.String)Z
10 v24 = hider.InstallService→uninstallapp(this, "hider")
11 v36 = hider.InstallService→execCommand1(p1, v33)
12 v33 = java.lang.StringBuilder→toString{v30}()
13 v30 = java.lang.StringBuilder→append{v26}(v21)
14 java.lang.StringBuilder→<init>{v26}("pm install -r ")
15 v26 = new java.lang.StringBuilder
16 java.io.File→<init>{v21}(v23, p2)
17 v21 = new java.io.File
18 v23 = hider.InstallService→getFilesDir{this}()
19 Entry hider.InstallService.runRootCommand1 (android.content.Context,
       java.lang.String)Z
20 v44 = hider.InstallService→runRootCommand1{this}(this, "newapp.apk")
21 v33 = hider.InstallService→runRootCommand1{this}(this, "testnew.apk"
       )
```

The rendering module receives these slices for each execution path and transforms them into structured and simplified code (see Output 3.6). The result precisely shows that the malware uses this code segment to both uninstall itself from the system (line 1) and to install its supplemental apk file, either named *newapp.apk* or *testnew.apk*. Manual analysis of samples of this family revealed that each sample only contains one of these supplemental apks (presumably to evade signature-based malware detection mechanisms).

3.7 Discussion

Our value analysis performs a sequence of optimizations on slices to minimize the number of instructions and to increase the expressiveness of the individual instructions while preserving the semantics. One step comprises encoding domain knowledge of Java API methods, e.g., to simplify a series of StringBuilder operations. This optimization currently only works if rules exist for all operations of a given call chain. Moreover, the domain knowledge, only keeps minimal state to reason about the return value given a

Output 3.5: Control-flow-ordered slices for each of the three execution paths from Output 3.4

```
1  Execution Path P1:
2  v10 = java.lang.Runtime→exec{v7}(p2)
3  v7 = java.lang.Runtime→getRuntime()
4  Entry hider.InstallService.execCommand1 (android.content.Context,java
       .lang.String)Z
5  v27 = hider.InstallService→execCommand1(p1, v24)
6  v24 = java.lang.StringBuilder→toString{v21}()
7  v21 = java.lang.StringBuilder→append{v17}(p2)
8  java.lang.StringBuilder→<init>{v17}("pm uninstall ")
9  v17 = new java.lang.StringBuilder
10 Entry hider.InstallService.uninstallapp (android.content.Context,java
       .lang.String)Z
11 v24 = hider.InstallService→uninstallapp(this, "hider")
12
13 Execution Path P2:
14 v10 = java.lang.Runtime→exec{v7}(p2)
15 v7 = java.lang.Runtime→getRuntime()
16 Entry hider.InstallService.execCommand1 (android.content.Context,java
       .lang.String)Z
17 v36 = hider.InstallService→execCommand1(p1, v33)
18 v33 = java.lang.StringBuilder→toString{v30}()
19 v30 = java.lang.StringBuilder→append{v26}(v21)
20 java.lang.StringBuilder→<init>{v26}("pm install -r ")
21 v26 = new java.lang.StringBuilder
22 java.io.File→<init>{v21}(v23, p2)
23 v21 = new java.io.File
24 v23 = hider.InstallService→getFilesDir{this}()
25 Entry hider.InstallService.runRootCommand1 (android.content.Context,
       java.lang.String)Z
26 v44 = hider.InstallService→runRootCommand1{this}(this, "newapp.apk")
27
28 Execution Path P3:
29 v10 = java.lang.Runtime→exec{v7}(p2)
30 v7 = java.lang.Runtime→getRuntime()
31 Entry hider.InstallService.execCommand1 (android.content.Context,java
       .lang.String)Z
32 v36 = hider.InstallService→execCommand1(p1, v33)
33 v33 = java.lang.StringBuilder→toString{v30}()
34 v30 = java.lang.StringBuilder→append{v26}(v21)
35 java.lang.StringBuilder→<init>{v26}("pm install -r ")
36 v26 = new java.lang.StringBuilder
37 java.io.File→<init>{v21}(v23, p2)
38 v21 = new java.io.File
39 v23 = hider.InstallService→getFilesDir{this}()
40 Entry hider.InstallService.runRootCommand1 (android.content.Context,
       java.lang.String)Z
41 v33 = hider.InstallService→runRootCommand1{this}(this, "testnew.apk"
       )
```

Output 3.6: Rendered optimized code for Output 3.4

```
1 P1: java.lang.Runtime→getRuntime()→exec("pm uninstall hider")
2
3 P2: x1 = java.io.File→<init>(hider.InstallService→getFilesDir(), "↷
      newapp.apk")
4     java.lang.Runtime→getRuntime()→exec("pm install -r "+ x1)
5
6 P3: x1 = java.io.File→<init>(hider.InstallService→getFilesDir(), "↷
      testnew.apk")
7     java.lang.Runtime→getRuntime()→exec("pm install -r "+ x1)
```

number of input arguments. A more comprehensive domain-specific language is required to also cover complex APIs which might require a model of the class internals, for instance, as a state machine. Automatically deriving such expert knowledge from the framework code is an interesting topic for future work.

Reassembling execution paths during optimization is a valuable step to increase precision in specific scenarios (see Section 3.4) by enabling automatic checks on values rather than API methods. However, this path retargeting comes with certain limitations. As described in Section 3.6.2, in about 20% of the sinks analyzed the path recovery either hit the 3-minute threshold or the maximum of 32 paths. Sinks with a large set of instructions originating from different classes/methods and with a high number of loops or branching constructs may quickly lead to a path explosion. This is amplified by the fact that our path recovery solely works on complete slices, i.e. typically inter-procedurally. Being able to selectively retarget simpler cases intra-procedurally may lead to a better optimization rate. Due to the general undecidability of loop conditions, we cannot statically infer the concrete number of iterations. Internally, loops constitute an if-conditional with a jump instruction, i.e. two execution paths. We soundly capture data-dependencies through both execution paths, but we do not strive for approximating the exact loop iterations. In this cases, we still detect sensitive flows through loops, but we cannot determine exact values. Similarly, we flatten recursive calls.

Although the slicer is capable of capturing control-dependencies, our use-def tracker currently does not handle path conditions. This implies that R-Droid cannot detect implicit flows via conditions. Implementing this feature is part of future work, as it also improves code understanding for a human analyst, i.e. when for any execution path there is a set of trigger conditions.

The code rendering module implements a simple means of restructuring slice instructions to improve readability. However, in complex scenarios, grouping instructions by call chains is not optimal and does not allow a fluent code reading. A different approach to presenting slice instructions includes disassemblers. Given a mapping of WALA IR instructions back to dex bytecode instructions and a mapping of bytecode instructions and disassembled code, one could try to show only the disassembled JAVA code that is represented in the slice. One particular challenge is to provide enough context to understand the code. A solution to this could be executable slices [26], that even provide

enough context to compile the set of instructions again to a runnable program.

A widely disregarded aspect in mobile privacy leak analysis is considering sanitizers to reduce the number of false positives. In the presence of a data-dependency between a sensitive source and a sink, a sanitizer method could have been applied to protect user data or to infer a less-sensitive data token. Web application vulnerability scanners that test for injection attacks like SQL injections typically define a set of user-input sanitizers such as real_escape_string or htmlspecialchars to classify problematic and non-problematic data-flows. However, defining proper sanitization methods for different source categories in Android is more challenging. A hashed unique identifier that is sent out, still serves a unique identifier, although the real value has been blinded. Thus, for each source category, one would have to define privacy-preserving transformation methods. Such a set of sanitizers could then be used to distinguish leaking and privacy-preserving code.

3.8 Related Work

Improving Android's security has received a lot of attention in the security community. Over the last years, a larger number of analysis frameworks was published with a focus on information-flow aspects such sensitive data/user input leakage. Table 3.4 shows a high-level feature comparison of static analysis frameworks on Android sorted by publication year in ascending order. For the sake of simplicity, we distinguish feature support in three categories: ✓ = comprehensive/sophisticated, ● = basic, and ✘ = no support.

Most related work [32, 80, 94, 144, 63] performs a data-dependency analysis on top of an application (lifecycle) model. While this is sufficient for binary assessments, i.e. is there a dependency between a source and a sink statement, more complex problems like reconstructing values/strings passed to an API call require a detailed list of dependent statements. Moreover, such data does not necessarily have to be derived from an API call (see premium SMS example in Section 3.4). Similar use-cases such as API (mis-)use [112, 106, 54, 52] in which the concrete (string) data needs to be assessed cannot efficiently be processed by forward approaches since *any* string has to be marked as a source. In contrast, our backward slicing approach fulfills the requirements to conceptually handle all of these problems.

Table 3.4 shows an evolution in terms of supported features. The foundation of a static analysis is a comprehensive data model that precisely approximates Android's runtime behavior. *Chex* [94] was the first tool to take the different types of application entry points into account. *FlowDroid* [19] improved on this by introducing accurate per-component lifecycle models that were also adopted by *AmanDroid* [144]. R-Droid follows prior work and additionally adds models for frequently used classes to further refine the static lifecycle model.To keep the analysis complexity tractable almost all approaches manually modeled parts of the semantics of the framework API instead of adding the full framework code base to the analysis scope. *DroidSafe* [63] proposed a new technique to abstract from the framework complexity while still keeping all data-

		SCanDroid [32]	Scandal [80]	AndroidLeaks [59]	LeakMiner [151]	Chex [94]	FlowDroid [19]	AmanDroid [144]	DroidSafe [63]	R-Droid
Approach	Framework used	WALA	custom	WALA	Soot	WALA	Soot	custom	Soot	JOANA
	Fwd/Bwd analysis	—	Fwd	Fwd	Fwd	—	Fwd	Fwd	Fwd	Bwd
	Analysis approach	DD	DD	Slicing	Tainting	DD	Tainting	DD	DD	Slicing
	String analysis	●	●	✘	✘	✘	✘	●	●	✓
Features	Apk parsing	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Lifecycle model	●	●	●	●	●	✓	✓	●	✓
	Framework model	●	●	●	●	●	●	●	✓	✓
	ICC support	✓	✘	✘	✘	✘	✓(*)	✓	✓	✘
	Concurrency	●	✘	✘	✘	✓	✓	✘	✓	✓
	Reflection	✘	●	✘	✘	✘	●	✘	✘	✘
	Native code	✘	✘	✘	✘	✘	✘	✘	✘	✘

✓ = comprehensive/sophisticated support, ● = basic support, ✘ = no support, DD = Data-Dependency, (*) ICC support was added through IccTA [84].

Table 3.4: High-level feature comparison of static analysis frameworks and extensions for Android. Frameworks sorted by publication year in ascending order.

dependencies. R-Droid adopts this model to not rely on incomplete knowledge about the framework API. A dedicated line of work focused on Android's inter-component communication (ICC) either as standalone approach [105, 104] or as part of an analysis framework [32, 84, 144]. While tracking data-flows across components clearly increases the precision of the overall analysis, our privacy leak evaluation (cf., Section 3.6.4) showed that such sensitive flows rarely occur in real-world apps.

String Analysis. Most of the aforementioned work on API usage analysis as well as first works on ICC resolution [105] resort to rather limited constant propagation approaches tailored to the specific use-case. *SCanDroid* [32] resolves ICC receivers by implementing a constraint system to track values across instructions. String values are approximated by constructing a subgraph that includes StringBuilder operations. Flow solver algorithms then compute feasible string prefixes that flow into ICC-related calls. Christensen *et al.* [36] propose the Java String Analyzer (*JSA*) to statically check the syntax of dynamically generated expressions like SQL queries. Dedicated flow-graphs for string operations are translated into context-free grammars to infer possible string values. *DroidSafe* uses *JSA* to resolve ICC. The *IC3* project [104] is the closest related approach to our value analysis. They employ context-sensitive, inter-procedural composite constant propagation that can handle field correlations of complex objects. To describe the semantics of the Android API (we refer to this as expert knowledge) they devised a declarative language and use a constraint solver to output possible string values. Similarly, our value analysis uses inferred expert knowledge, but, in contrast, our approach is also object- and path-sensitive. Since our multi-stage value analysis is not restricted to strings, even complex propagation problems such as adding values to arrays/lists with subsequent retrieval via index calculations is possible.

Statically resolving string values is also highly relevant beyond Android. Automata-based approaches for string analysis exist to verify SQL expressions [119] or for finding string-related security vulnerabilities in PHP programs [153, 154]. Tateishi *et al.* [130] apply path- and index-sensitive string analysis to verify Cross-Site-Scripting (XSS) sanitizers for web applications based on monadic second-order logic (M2L). They verify that generated strings satisfy a given constraint rather than assembling concrete values like R-Droid does. String solvers have recently also been integrated into SMT solvers [137, 87, 157]. These solvers determine whether a certain string can be produced by a program (e.g. whether a XSS attack is possible). In contrast, R-Droid determines *which* concrete strings can be created in an app.

Support for Manual Security Audits Providing assistance to an human auditor to assess the produced output of an analysis has received only little attention so far. While refining analysis techniques to generate more accurate results is beneficial for manually validating and assessing the output, this just constitutes an essential but insufficient step towards an efficient reviewing process. One approach to tackle this problem is reducing the slice output by filtering nodes that are irrelevant to the human auditor. Krinke proposes slicing with barriers [82], in which the basic idea is to stop the slicer at code

boundaries that are known to be safe or irrelevant for the analyst, e.g. utility code in a library. Similarly, we filter instructions within the Android API, as our set of sensitive sources and sinks is defined via Android API signatures. Thomé *et al.* use JOANA to generate security slices in Java web applications to detect injection attacks [135]. They propose different pruning techniques to minimize slices, such as filtering code from known libraries, or vulnerabilities that can be fixed automatically. Moreover, they define sanitizer functions for user inputs, to remove false positives whenever such sanitizers are applied to data that hasn't reached a sensitive sink. Defining input sanitizers for web applications is, however, simpler than source sanitizers in Android (see Section 3.7). In contrast to such filtering-only approaches, R-Droid applies a comprehensive value analysis to further produce semantically-equivalent, small-sized slices. Our slice rendering module subsequently generates more readable, code-like output to facilitate a manual investigation.

3.9 Conclusion

We presented a novel slice optimization approach as post-processing to standard slicing techniques. As part of this approach, we devised a comprehensive value analysis to retarget strings and values beyond constants. This allows a larger number of security and privacy-related use-cases, such as various API (mis-) use analyses, to be assessed in an automatic way. Moreover, the concise output of R-Droid supports experts in understanding app functionality and in manually reviewing the results, a mandatory task for any static analysis.

A yet unsolved, but important, problem is being able to distinguish app developer code and code from third-party libraries that are statically included. In Chapter 5 we address the problem of attributing security and privacy-related problems to the correct entity, i.e. app developer or libraries, by proposing a method to detect third-party libraries in app binaries even in presence of common code obfuscations. This approach has another advantage that security analyses, such as privacy-leak detection, can be pre-computed for libraries. Upon detection of libraries, the results can then be used directly, without the need to include the library code in the actual analysis.

4

axplorer

On Demystifying the Android Application Framework:

Re-Visiting Android Permission Specification Analysis

4.1 Motivation

In contrast to the application layer, Android's application framework's internals and their influence on the platform security and user privacy are still largely a black box for us. In this work, we establish a static runtime model of the application framework in order to study its internals and provide the first high-level classification of the framework's protected resources. We thereby uncover design patterns that differ highly from the runtime model at the application layer. We demonstrate the benefits of our insights for security-focused analysis of the framework by re-visiting the important use-case of mapping Android permissions to framework/SDK API methods. We, in particular, present a novel mapping based on our findings that significantly improves on prior results in this area that were established based on insufficient knowledge about the framework's internals. Moreover, we introduce the concept of permission locality to show that although framework services follow the principle of separation of duty, the accompanying permission checks to guard sensitive operations violate it.

4.2 Problem Description

Android's application framework—i.e., the middleware code that implements the bulk of the Android SDK on top of which Android apps are developed—is responsible for the enforcement of Android's permission-based privilege model and as such is also a popular subject of recent research on security extensions to the Android OS. These extensions provide various security enhancements to Android's security, ranging from improving protection of the user's privacy [103, 162], to establishing domain isolation [107, 30], to enabling extensible access control [71, 21].

Android's permission model and its security extensions are currently designed and implemented as best-effort approaches. As such they have raised questions about the efficacy, consistency, or completeness [4] of the policy enforcement. Past research has shown that even the best-efforts of experienced researchers and developers working in this environment introduce potentially exploitable errors [51, 155, 125, 120]. In light of the framework size (i.e., millions of lines of code) and based on past experience [51, 155, 57, 120, 129], static analysis promises to be a suitable and effective approach to (help to) answer those questions and hence to demystify the application framework from a security perspective. Unfortunately, on Android, the technical peculiarities of the framework impinging on the analysis of the same have not been investigated thoroughly. As a consequence, past attempts on analyzing the framework had to resort to simple static analysis techniques [20]—which we will show as being insufficient for precise results—or resort to heuristics [120].

In order to improve on this situation and to raise efficiency of static analysis of the Android application framework, one is confronted with open questions on *how to enable more precise static analysis of the framework's codebase*: where to start the analysis (i.e., what is the publicly exposed functionality)? Where to end the analysis (i.e., what are

the data and control flow sinks)? Are there particular design patterns of the framework runtime model that impede or prevent a static analysis? For the Android application layer, those questions have been addressed in a large body of literature. Thanks to those works, the community has a solid understanding of the sinks and sources of security- and privacy-critical flows within apps (e.g., well-known Android SDK methods) and a dedicated line of work further addressed various challenges that the Android *application* runtime model poses for precise analysis (e.g., inter-component communication [105, 144, 84, 104] or modelling the Android app life-cycle[94, 19]). Together those results form a strong foundation on which effective security- and privacy-oriented analysis is built upon. In contrast to the app layer, for the application framework we have an intuitive understanding of what constitutes its entry points, but no in-depth technical knowledge has been established on the runtime model, and almost no insights exist on what forms the security and privacy relevant targets of those flows (i.e., what technically forms the sinks or "protected resources").

4.3 Contribution

This work contributes to the demystification of the application framework from a security perspective by addressing technical questions of the underlying problem on *how* to statically analyze the framework's code base. Similar to the development of application layer analyses, we envision that our results contribute some of the first results to a growing knowledge base that helps future analyses to gain a deeper understanding of the application framework and its security challenges.

How to Statically Analyze the Application Framework. We present a systematic top-down approach, starting at the framework's entry points, that establishes knowledge and solutions about analyzing the control and data flows within the framework and that makes a first technical classification of the security and privacy relevant targets (or resources) of those flows. The task of establishing a precise static runtime model of the framework was impeded by the absence of any prior knowledge about framework internals beyond black-box observations at the framework's documented API and manual analysis of code fragments. Hence we generate this model from scratch by leveraging existing results on statically analyzing Android's application layer at the framework layer. The major conceptual problem was that the design patterns of the framework strongly differ from the patterns that had been previously encountered and studied at the application layer. Consequently, we devised a static analysis approach that systematically encompasses all framework peculiarities while maintaining a reasonable runtime. As result of this overall process, we have established a dedicated knowledge base that subsequent analyses involving the application framework can be soundly based upon.

AXPLORER Tool and Evaluation. Unifying the lessons learned above, we have built an Android application framework analysis tool, called AXPLORER. We evaluate AXPLORER

on four different Android versions—v4.1.1 (API level 16), v4.2.2 (17), v4.4.4 (19), and v5.1 (22)—validate our new insights and demonstrate how specialized framework analyses, such as message-based IPC analysis and framework component interconnection analysis, can be used to speed up subsequent analysis runs (e.g. security analyses) by 75% without having to sacrifice precision. As an additional benefit, the resulting output can be used by independent work *as is* to create a precise static runtime model of the framework without the need to re-implement the complex IPC analysis.

Android Permission Analysis. Finally, to demonstrate the benefits of our insights for security analysis of the framework, we conduct an Android permission analysis. In particular, we re-visit the challenge of creating a permission map for the framework/SDK API. In the past, this problem has been tackled [113, 20] without our new insights in the peculiarities of the framework runtime model, and our re-evaluation of the framework permission map reveals discrepancies that call the validity of prior results into question. Using AXPLORER, we create a new permission map that improves upon related work in terms of precision. Moreover, we introduce a new aspect of permission analysis, permission locality, by investigating which framework components enforce a particular permission. We found permissions that are checked in up to 10 distinct and not necessarily closely related components. This indicates a violation of the separation of duty principle and can impede a comprehensive understanding of the permission enforcement. As a side-effect, we can leverage our set of permission-protected APIs to refine commonly used source-sink lists for app analysis that are generated by approaches built on coarse-grained approximations of the framework model.

4.4 Technical Problem Description and Approach

In contrast to the various related works on static analysis at the application level, there is no existing prior work on in-depth analysis of the application framework. Moreover, as the architecture of the framework fundamentally differs from the architecture of applications, open questions have to be answered first to be able to conduct in-depth static analysis of the framework. For instance, *"what are the entry points to the application framework?"* or *"how to establish a static runtime model of the framework's control flows?"* In the following, we identify challenges that arise for static analyses at framework level and present a systematic, top-down approach to cope with these problems (an implementation of our approach is presented in Section 4.5). Solving the discussed challenges lays the foundation on which a wide range of security analyses of the application framework can be constructed, from which we (re-)visit the use-case of permission analysis in Section 4.7.

4.4.1 Defining Framework Entry Points

The first question to be answered is how to identify and select the starting points for the framework analysis? At application level this has already been studied in depth [94, 19,

32, 59, 151]. From a high-level view, most approaches parse the declared components from the application manifest and determine the components as well as dynamically registered callbacks as entry points; or they build component/application life-cycle models with a single main entry method.

> **Challenge:** *The framework model is conceptually different from the application layer and existing approaches for application layer analysis do not apply in a framework analysis. Instead one has to identify the framework API methods that are exposed to app developers as analysis entry points.*

To identify the entry point methods, we have to locate the relevant framework entry point classes. Starting with the official API of the Android SDK (e.g., Managers in Figure 2.1) is not reliable as there are no means to prevent an app developer from bypassing the SDK by immediately communicating with the framework services or by using reflection to access hidden API methods of the SDK. Consequently, we do not consider the API calls within the SDK as entry points but instead the framework classes that are entry points for accessing framework functionality (i.e., framework classes that are being called by the SDK, see Figure 2.1). We exclude entry points that are not accessible by app developers, such as Zygote, service manager, or the property service, which are under special protection (e.g., SELinux [131]) and will not accept commands by third-party apps that have tangible side-effects on the system or other apps. This restriction is in accordance with the design of existing Android security extensions, which exclusively focus on the exported functionality of the app framework (e.g., the framework's bound services).

Inter-component communication in Android is, by design, based on Binder IPC and, thus, framework classes have to expose functionality via Binder interfaces to the application layer. To this end, interfaces must be derived from IInterface, the base class for Binder interfaces. Binder interfaces might automatically be generated by AIDL, in this case, the entry point classes extend the auto-generated Stub class, or in case of Binder interfaces that are not generated by AIDL, a custom Binder implementation like ActivityManagerNative[1] has to be provided, which in turn is extended by the entry point classes. These class relationships can be resolved via a class hierarchy analysis (CHA) to determine the set of all entry classes. Besides bound services this also includes callback and event listener classes that expose an implementable interface to app developers. Hence, we define entry points (EP) as the public methods of framework classes that are exposed via a Binder interface. In addition, permission-protected entry points (PPEP) are defined as entry points from which a permission check is control-flow reachable.

4.4.2 Building a Static Runtime Model

> **Challenge:** *Generating a static model that approximates the runtime behavior of the application framework again strongly differs from the problems that arise at application level where the component life-cycles are mimicked to approximate runtime behavior.*

[1] By convention non-generated class names end with *Native*.

> *The bound services—as entry points to the framework—might be queried simultaneously from multiple clients (apps) via IPC and hence have to handle multi-threading to ensure responsiveness of the framework. In contrast to the application layer at which utility classes like* AsyncTask *are used for threading, we discovered that the framework services make intensive use of more generic but also more complex threading mechanisms like* Handler*,* AsyncChannel*, and* StateMachines*. Disregarding these concurrency patterns results in imprecise data models that cause a high number of false positives during a framework analysis.*

In the following, we provide technical background for those asynchronicity patterns and explain how to statically model them correctly.

4.4.2.1 Handler

The class android.os.Handler provides a mechanism for reacting to messages or submitting Java Runnable objects for execution on a (potentially remote) thread. Handlers either schedule the processing of a message or the execution of a Runnable at some point in the future or process a message/Runnable in a separate thread.

To illustrate the Handler mechanism, consider the example shown in Listing 4.1. It includes the relevant parts of the framework class com.android.server.BluetoothManagerService. When the service is constructed, it instantiates a HandlerThread object (line 6), a traditional Thread object associated with a Looper. The purpose of the Looper class is to sequentially process the messages in a message queue. At line 8, the class-specific BluetoothHandler object is created and associated with the newly created Looper from the HandlerThread. This allows messages sent to the BluetoothHandler to be pushed to the message queue for this Looper. Methods enable and disable can be called by applications via RPC on IBluetoothManager to turn the Bluetooth functionality on or off. Method enable sends a message with code MESSAGE_ENABLE to the BluetoothHandler (line 12). When the associated Looper instance processes the message, it calls method handleMessage in the BluetoothHandler (line 20), which then processes the request.

Statically resolving message-based IPC, requires overcoming several challenges. First, the runtime type of the Handler instance has to be inferred to determine the concrete handleMessage method of the receiving class that processes the message. Second, to add precision to the analysis, it is best to make it locally path-sensitive by inferring the possible message codes of the arguments to sendMessage methods. For the example presented in Listing 4.1, this enables the analysis to be limited to the feasible paths for a given message in the switch statement at line 21. While it is possible to perform the analysis without this information, doing so results in a significant loss of precision and, thus, an increase in the number of false positives, which may distort the results of security analyses built on top. In light of the prevalence of the Handler pattern, this loss of precision is not an acceptable solution. Finally, since messages can also be associated with runnable tasks instead of message codes, the concrete Runnable types, associated with each message, have to be inferred to determine the executed code when such a message is processed.

Listing 4.1: Bluetooth Handler in the Bluetooth manager service. Code was simplified for readabililty.

```
1  class BluetoothManagerService {
2    private HandlerThread mThread;
3    private BluetoothHandler mHandler;
4
5    public BluetoothManagerService() {
6      mThread = new HandlerThread("BluetoothManager");
7      mThread.start();
8      mHandler = new BluetoothHandler(mThread.getLooper());
9    }
10   public void enable() {
11     Message msg = mHandler.obtainMessage(MESSAGE_ENABLE);
12     mHandler.sendMessage(msg);
13   }
14   public void disable() {
15     Message msg = mHandler.obtainMessage(MESSAGE_DISABLE);
16     mHandler.sendMessage(msg);
17   }
18
19   class BluetoothHandler extends Handler {
20     public void handleMessage(Message msg) {
21       switch (msg.what) {
22         case MESSAGE_ENABLE:
23           // process enable message
24           break;
25         case MESSAGE_DISABLE:
26           // process disable message
27           break;
28         // Other cases.
29   }}}
30 }
```

4.4.2.2 AsyncChannel

Closely related to Handlers, com.android.internal.util.AsyncChannel implements a bi-directional channel between two Handler objects. It provides its own sendMessage and replyToMessage methods, both of which delegate to the sendMessage methods in its associated Handler. In order to precisely model AsyncChannel objects, it is necessary to infer the types of the sender/receiver Handler objects. Similarly to Handlers, path-sensitivity should be added to the analysis by inferring the message codes that are sent through the channel.

4.4.2.3 StateMachine

Building on the Handler concept, the com.android.internal.util.StateMachine class models complex subsystems such as the DHCP client or the WiFi connectivity manager. This class allows processing of messages depending on the current state of the modeled system. It effectively constitutes a hierarchical state machine in which messages cause

state transitions. States are organized in a hierarchical manner, such that parent states may process messages that are not handled by child states.

State objects may optionally implement an enter and exit method that are called when the state is entered or exited, respectively. When performing a transition from a source to a destination state, the state machine first determines the lowest common parent for the source and destination states. It then proceeds to call exit methods of the states in order from the source to the lowest common parent in the hierarchy. Finally, it calls the enter methods of the destination's ancestors starting at the lowest common parent and ending at the destination state.

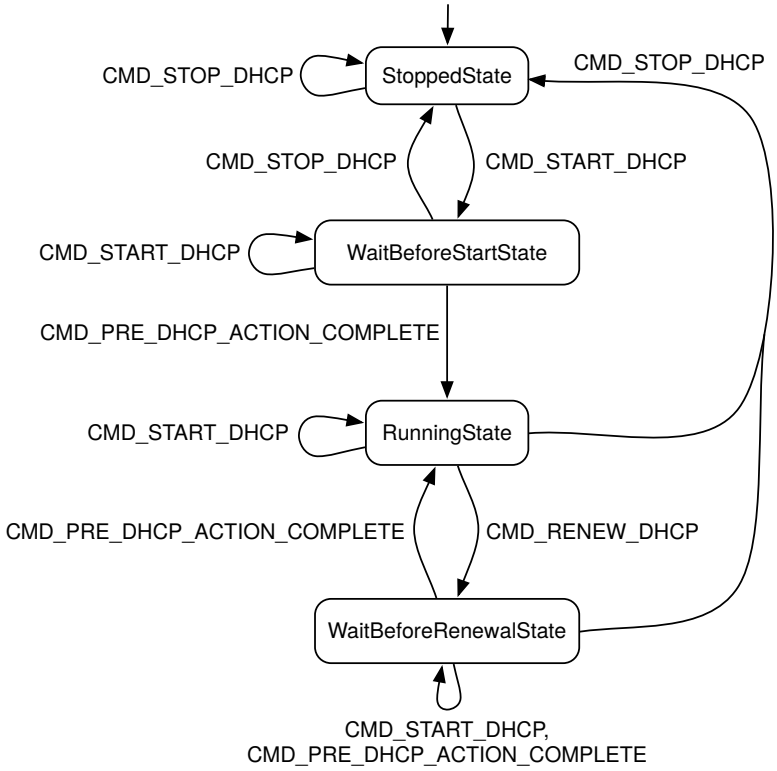


Figure 4.1: Simplified DHCP state machine from class android.net.DhcpStateMachine. Omitted commands do not cause state transitions. States provided by the default implementation (halting and quitting) are not shown.

Figure 4.1 shows the state machine used for the DHCP system in the Android framework. It comprises states StoppedState, WaitBeforeStartState, RunningState and WaitBeforeRenewalState. The exact meaning of each state and transition is not relevant for this explanation. This state machine is initially in state StoppedState. The state machine can then receive messages such as CMD_START_DHCP through a call to one of its sendMessage methods. This causes the message to be transmitted to an internal Handler object. The handleMessage method of the Handler then dispatches the message to the current State object, which processes it.

In order to precisely model state machines, several challenges must be addressed. First, the subtype of the state machine itself must be inferred, including all states and possible transitions. Second, the hierarchy of the states must be inferred, in order to know which enter and exit state methods are called upon state transitions. This information is also required to determine the set of states that may handle a given message. Third, for eliminating further false positives, one needs to infer the possible states for any given program location at which interaction with the state machine occurs.

4.4.3 Identifying Protected Resources

While the previous sections describe how static analysis of the Android application framework code base can be enabled, we now classify the resources inside the application framework that actually have to be protected. Unfortunately, there is a lack of consensus in the community on what constitutes a security-sensitive resource/operation [18, 57] and no one-size-fits-all definition exists as the concrete definition depends on various aspects like operating system, programming language, or even the domain. To avoid ambiguities on what we denote as a protected resource in the following, we note that protected resources for us are security sensitive operations that have a tangible side effect on the system state or use of privacy.

> **Challenge:** *Defining the security-relevant resources is, in contrast to entry points, more challenging. For privacy leak analysis at application-level, there is a well-defined list of API methods that can be classified as sinks. Since the analysis now shifts into the API methods of the framework, it is unclear what kind of resources are protected by Android's permissions and can, thus, be used as sinks for security analysis within the framework.*

To create a first high-level taxonomy of protected resources that can help to automatically discover such resources, we first have to create a ground truth about what technically forms a protected resource. To this end, we manually investigated control flows of a number of identified PPEP in the framework's source code. Here, we make the assumption that every existing permission check within the application framework indeed controls access to at least one security- or privacy-critical system resource. Checks are usually located at the very beginning of PPEP so that any subsequent operation is indeed authorized. Using expert knowledge in combination with descriptions of expected side effects from the Android documentation, we identify and annotate relevant statements that modify the service and/or system state. To avoid a potential bias in the types of protected resources, we chose entry points from eight different entry classes. To cover a variety of distinct cases, we based our selection on the available information such as return value, number/type of EP input arguments, or number/type of permission checks collected during the entry point discovery. After manually investigating flows from 35 entry points, distinct repetitive patterns for protected resources appeared across different control flows, which we summarized in a taxonomy of the high-level protected resource types.

4.4.3.1 Taxonomy of Protected Resources

Figure 4.2 presents our high-level taxonomy of the protected resource types. In contrast to work at application-level that disregards field instructions [18], we found that *field update* instructions are highly relevant in the context of the framework and in fact are the most prevalent type of protected resources that we discovered. Relevant *method invocations* can be further sub-classified into native method calls (e.g. for file system access or modification of device audio settings) and broadcast sender. We consider native method calls generally as protected resources since distinguishing (non-)security-relevant native calls would require a dedicated analysis for the native code, which is currently a general, open problem for the community and out of scope for this work. Broadcast senders are protected resources as they can potentially cause side-effects in receivers registered by the system or apps. However, this is statically unresolvable, as the concrete side-effects strongly depend on the current system configuration, e.g. on the installed apps and the set of active broadcast listeners. We consider non-void *return values* of security-sensitive entry methods as protected resource. Returned objects of such methods constitute sensitive data, e.g., a list of WiFi connections. Return values of primitive types int or boolean may constitute sensitive values like for the method isMultiCastEnabled of the WifiService or some status/error code in method enableNetwork of the same service. We also found cases in which a *throw RuntimeException* (RTE) has to be considered as a protected resource. For instance, in the crash method of the PowerManagerService, that requires the caller to hold the reboot permission, an RTE (intentionally) causes the runtime to crash and the device to reboot in consequence.

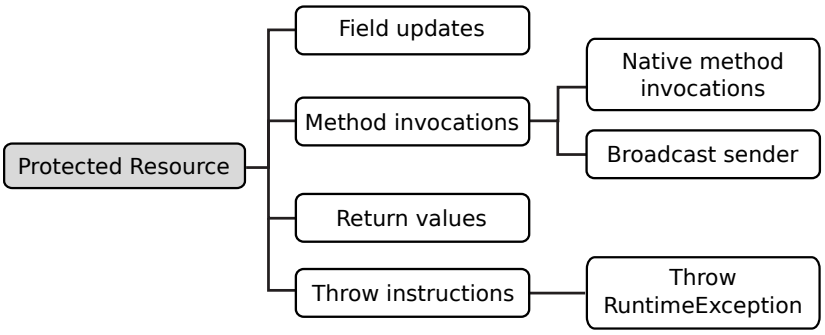


Figure 4.2: High-level taxonomy of protected resource operation types.

Refining Field-update Resource Types The established taxonomy of high-level resource types constitutes a simple, yet effective means of classifying instructions into security-relevant or not. The set of aggregated instructions, however, is still insufficiently precise and likely results in an over-approximation, i.e. superset, of the actual protected resources. This is particularly true for field updates that have a share of about 75% of all protected resources. Being a human expert, it is already challenging to decide whether a given field update causes side-effects to the service or even system state. Trying to fully automate this process is generally not possible, but we can devise heuristics to approximate human expertise.

A first simple heuristic to eliminate false positives targets updates of a this reference within constructors. As there is no prior state for this object, such updates are irrelevant in this context. By discarding them, we do not miss any information, because adding a new object to some container or re-assigning it to some other field will result in another protected instruction in the superset. Hence, field instructions in constructors are false positives and can be filtered from the superset.

The present approach will also flag field updates identified within classes of the Java package as protected resources. If for instance, a framework service contains a member field of a Java container class and a new element is added, the control-flow slicer traverses the add method of this container class and finally marks the field update instruction on the container member array as protected resource. While this approach is technically correct, it has a severe drawback. By collecting the field update within the Java method the context will be eliminated, i.e., it is unclear which call originally triggered this update. This may cause false negatives when there are distinct control-flows originating from different code locations that converge into the same Java method.

To remedy this situation, we proceed as follows: Whenever a candidate field update instruction is found in a Java class, we abstain from flagging this instruction as a protected resource. Instead, we traverse the current stack trace backwards until we reach the first method call within the Android namespace, i.e. android, com.android, or dalvik. This method call is then marked as a protected resource of type *field update*, as it triggers the respective field instruction. This abstraction reconstructs the context and prevents false negatives in case different code locations converge into the same Java method.

4.4.3.2 Coverage of the Taxonomy

An inherent limitation of our taxonomy based on small-scaling manual analysis is, that there are no guarantees that corner cases are included in the current classification. To cover all corner cases in our taxonomy, a comprehensive manual analysis of the framework would be required, which would defeat the purpose of enabling a static analysis in the first place. We define a high-level taxonomy of protected resource (types) in the framework. Distilling a more refined set for security analyses is discussed separately in Section 4.8.

4.5 Implementation

We combined all aforementioned steps from Section 4.4 for analysis of an arbitrary framework version into a tool called Axplorer. We leverage the static analysis framework WALA [77], although our approach is equally applicable to other analysis frameworks such as Soot [126]. Additional code for realizing our approach comprises ≈15 kLOC of Java.

4.5.1 Call-graph Generation

For each identified entry class, we generate an inter-procedural call-graph (CG), including all reachable classes. As opposed to related approaches [20] that use class hierarchy analysis to generate low-precision call-graphs due to the overall framework complexity—Android version 4.2.2 already includes over 35,000 classes—we generate high-precision call-graphs with object-sensitive pointer resolution. For each virtual or interface invocation we infer the runtime type(s) and hence precisely connect the invocation to its target(s). Although the costs for the points-to computation are computationally expensive, the increased precision lowers the complexity of the overall call-graph, since we do not introduce imprecision by considering all subclasses of a virtual method call as potential receivers. Avoiding this imprecision in the call-graph also lowers the number of false positives. The complexity is further reduced by the design decision to not follow RPC calls to other entry classes. We complement the call-graph with message-based IPC edges during the control-flow slicing (see below).

4.5.2 Slicing & On-demand Msg-based IPC Resolution

We conduct a forward control-flow slice for each identified entry point method. The slicer stops at native methods, RPC invocations to classes other than the current one, and when the entry point method returns. During slicing, we perform an on-demand message/handler resolution to add message-based IPC edges to the call-graph, thus avoiding a huge computational overhead of computing all edges in advance when only a subset of them are required for analysis (e.g. if PPEP are analyzed only).

When the slicer reaches a sendMessage call, we infer the concrete handler type and add a call edge from the sendMessage call to the handleMessage method of the receiving handler. We augment this process with inter-procedural backwards slicing for two reasons: First, since existing type inference algorithms (like the ones implemented in WALA) work intra-procedurally, type inference fails if Handler objects are stored in fields whose declared field type is the Handler base class and not the concrete subtype. Using inter-procedural backwards slicing starting at the message-sending instruction, we obtain a more precise set of possible handler types in AXPLORER. Second, Messages are usually not constructed explicitly, but indirectly obtained via calls to Message.obtain or Handler.obtainMessage. These methods receive an integer value that constitutes a sender-defined message code that allows the recipient to identify the message type. To statically identify the message code, we compute a backwards slice starting from the message-sending instruction and check the resulting set of instructions for calls that construct/obtain a message. We then repeat this approach starting from the message obtain call to infer the concrete message code used to initialize the Message.

Handlers use switch statements to match the provided message code and to transfer control-flow to a specific basic block of the method's control-flow graph (cf. line 21 et seqq. in Listing 4.1). To avoid infeasible paths, we have to recreate path-sensitivity intra-procedurally and map the message code(s) to the individual execution path(s).

The control-flow slicer subsequently follows this specific execution path only to avoid a huge number of false positives. Runnable types in Handler.post calls are resolved in the same way and a call edge to the Runnable's run method is added. The approach slightly differs in case of StateMachines. Here, there is no single handleMessage function. Instead, each State implements its own processMessage function. In this case, we recreate path-sensitivity for each of these functions and delegate the control-flow to any matching switch statement.

4.6 Framework Complexity Analysis

We apply our gained insights from Section 4.4 to collect complexity information about the application framework. By doing this, we demonstrate how the analysis complexity can be held manageable to allow such in-depth analysis within a reasonable amount of time. Finally, we collect the framework's protected resources as denoted in our taxonomy and validate the results (a detailed discussion on how security analyses can benefit from this is given in Section 4.8). Using AXPLORER we analyze four different Android versions: 4.1.1 (API level 16), 4.2.2 (17), 4.4.4 (19), and 5.1 (22).

4.6.1 Handling Framework Complexity

Table 4.1 summarizes different complexity statistics generated for the four analyzed versions. Unsurprisingly, the complexity in terms of code increases with each version, whereas the gap to the most recent major version is significantly larger as between the minor version changes due to new features like Android TV. The entry class discovery algorithm identified between 242–383 entry classes of which ≈25% include at least one PPEP. The evaluation was conducted on a server with four Intel Xeon E5-4650L 2.60 GHz processors with 8 cores each and 768 GB RAM. Initial processing of the frameworks finished in reasonable time, ranging from 14–126 hours. Note that this computation has to be done only once per Android version and that there are no real-time constraints as, e.g., in application vetting. The most time-consuming task (about 90% of the overall time) was the generation of the high-precision call-graphs. In the following, we describe the use of entry-class interconnection and IPC analysis to speed up processing time without losing the precision of our data model.

4.6.1.1 Entry Class Interconnection

IPC-interfaces of framework entry classes are not only used by the application layer, but also by other framework services. Analyzing the communication behavior of entry classes does not only provide a deeper understanding of how the framework services are inter-connected but also facilitates analyses that rely on permission checks as security indicator (e.g., see Section 4.7). Exploiting the knowledge about which service EP triggers which RPCs along its control-flow enables pre-computation of execution path conditions and restricting the scope of a service analysis to only subsets of dependent

Android version	**4.1.1 (16)**		**4.2.2 (17)**		**4.4.4 (19)**		**5.1 (22)**	
# of classes	27,749		29,804		31,023		46,192	
- inner classes	14,784		15,936		17,525		28,933	
# of entry point classes	242		256		284		383	
- with ≥ 1 PPEP	64	(26.4%)	73	(28.5%)	75	(26.4%)	81	(21.2%)
# entry methods (EP)	2,583		2,734		2,861		3,225	
- thereof PPEP	863	(33.4%)	1,018	(37.2%)	1,227	(42.9%)	1,250	(38.8%)
- incl. IPC	328	(38.0%)	532	(52.2%)	518	(42.2%)	597	(47.8%)

Table 4.1: Comparing complexity measures for different Android versions (percentages relate to preceding line).

services rather than the entire framework (i.e, it allows to efficiently divide and conquer the framework analysis). In a post-processing step, the analysis results for distinct services can be stitched together at RPC boundaries.

A standard call graph provides information about the enclosing method/class of function calls. This information, however, is insufficient to reason about the interconnection of framework entry classes. Instead, the interesting information is the originating entry class that leads to an RPC rather than the actual class that encloses the RPC. To generate a map of RPC dependencies of entry classes, Axplorer records RPCs to other entry classes and maps them to the original entries during control-flow slicing. Figure 4.3 shows a subgraph of the overall RPC interconnections between flows from different entry classes on Android 5.1 that Axplorer generated. Nodes correspond to entry classes and are weighted by in-/out-degree, thus highlighting highly-dependent classes such as ActivityManagerService. The source of a directed edge is the originating entry class: the control-flow starts at an entry method of this class and at some point along the flow, not necessarily in the same class, an RPC to the class of the edge target node is invoked.

Across all four investigated Android versions there is a median number of three distinct RPC receivers per entry class in our map. The ActivityManagerService is an exceptional case where flows from its entry methods reach 36 different entry classes. These numbers emphasize that large parts of the framework are strongly connected and that detailed knowledge about the communication behavior greatly simplifies further framework analyses as explained above.

4.6.1.2 Message-based IPC Analysis

A precise model of the message sender to handler relations is crucial for the generation of a static runtime model of the framework with a low number of false connections. The last row in Table 4.1 shows the prevalence of the message sending pattern. Between 38–52% of PPEP include at least one message sending call. Across API levels, we found 300 (API 16) to over 500 (API 22) distinct message sender calls used within PPEPs. The evaluation of our IPC analysis showed that in 7% of all cases the message was sent to a StateMachine, and in 27% of all cases to a Handler. In the remaining 66% a Runnable

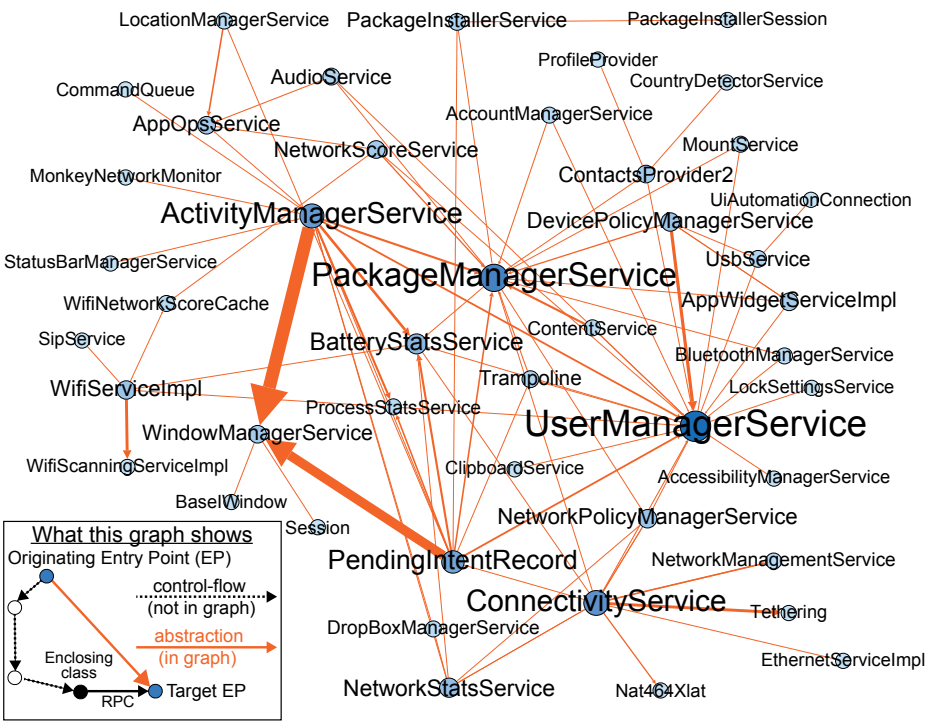


Figure 4.3: Subgraph of overall entry class interconnection via RPCs in Android 5.1. Directed edges show an RPC to an exposed IInterface method of target node. Nodes are weighted by in-/out-degree. Edges are weighted by number of RPCs.

was posted. This ratio remains approximately the same across all versions. Overall, our IPC analysis was able to fully resolve about 76% of all message sending instances, yielding a very valuable dataset of the message sender to handler relationships. Reasons for failed resolution are that either the runtime type of the Handler (81%) or Runnable (5%) instance could not correctly be inferred while in the remaining 13% of cases the message code could not be identified. The root cause of most of these failures is the missing/incomplete support of AsyncChannels and the Message.sendToTarget() API call. At the time of writing, this support is work-in-progress.

During our initial analysis run, AXPLORER records both an RPC-map per entry class as well as a list of resolved sender-to-handler relationships. This data is then re-applied as expert knowledge in subsequent analysis *re*-runs to significantly reduce the analysis runtime, e.g., for API level 17, the processing time drops by ~75% to about 7 hours. By publishing this data, we hope that independent analyses can equally benefit from this by removing the burden to re-implement a comparable IPC resolution algorithm.

4.6.1.3 Reflection

We analyzed reflection usage within framework code by counting the number of calls to methods within the java.lang.reflect package. The absolute numbers range from 89

(API 16) to 118 (API 22). Across API levels less than 50% targeted the Method class while the remaining calls were distributed among other reflection classes. In many cases reflection is used in utility or debug classes and we found only one entry class that makes use of reflection (ConnectivityService), but the respective method was removed in API level 20. In SDK code, the total numbers are slightly higher across API levels (115–288). However, the additional usage of reflection is mainly due to View/Widget classes. Overall, reflection is only rarely used in framework code and not used at all by main service components. Hence, not covering reflection during call graph construction has no significant impact on subsequent analysis results.

4.6.2 Android's Protected Resources

To validate our established taxonomy, we collect the protected resources for each Android version and classify them with respect to the taxonomy. Across versions, the total number ranges from 6,5k (API 16) to 10k (API 22). Although these numbers seem quite high at first glance, they are reasonable in relation to the overall size and complexity of the framework. AXPLORER recorded the context depth (in terms of method invocations) at which the protected resources were found. While for simple methods that include few (or even a single) resource, the call depth is lower than two, the median call depth ranges from 8–11 across Android versions. This emphasizes that approaches that resort to simple analysis techniques are not suitable to detect resources located deeper in the control-flow.

The relative distribution of resources per type is stable across all versions. We validated our statement of Section 4.4.3.1 that field update instructions are the most prevalent resource type (with a share of about 75%). They are followed by native method calls (about 21–23%), which are most frequently used as a gateway to the device hardware (e.g. file system, audio, NFC). There is a surprisingly low number of PPEP that return a protected value, the absolute number ranges from 51–69 entries. Another unexpected result is that runtime exceptions (RTE) occur with a frequency that is about as high as protected broadcast senders. Besides the already mentioned example within the PowerManagerService, we found occurrences in UI widget classes and even in the default XML parsing library on Android. Table 4.2 provides absolute numbers of protected resources by API version and the distribution of resource types.

Android version	**4.1.1 (16)**		**4.2.2 (17)**		**4.4.4 (19)**		**5.1 (22)**	
# of protected resources	6,490		6,969		7,488		10,044	
- field updates	4,891	(75.36%)	5,268	(75.59%)	5,675	(75.79%)	7,520	(74.87%)
- native method calls	1,433	(22.08%)	1,499	(21.51%)	1,643	(21.94%)	2,305	(22.95%)
- return value	51	(0.79%)	60	(0.86%)	54	(0.72%)	69	(0.69%)
- broadcast sender	65	(1.00%)	72	(1.03%)	53	(0.71%)	78	(0.78%)
- throw RTE	50	(0.77%)	70	(1.01%)	63	(0.84%)	72	(0.71%)
Median context depth	8		9		11		9	

Table 4.2: Numbers on protected resources by type and Android version.

4.6.2.1 Manual Investigation of RTE

Due to the surprisingly high number of runtime exceptions and the fact that uncaught RTEs might potentially crash the system, we manually investigated the corresponding code locations to better understand their security implications. The source code analysis revealed that these instructions reside both in Android's code base and in external library code. Android, similar to other operating systems, relies on external libraries for specific tasks, such as XMLPullParser, Bouncy Castle, and J-SIP. Integrating library code into such a complex system usually raises the question of how to handle unchecked exceptions that are thrown by library functions. Either the library code is patched, which might be a tedious maintenance task, or exceptions are caught at caller site. Although the latter case, that is adopted by Android, is considered bad practice in case of runtime exceptions, using generic catch handlers around library call sites is the easiest and most reliable approach in this situation.

Runtime exceptions in Android code are thrown to indicate precondition violations, like crucial class fields being null, unrecoverable IO errors, or security violations. The implications of a runtime exception differ with respect to the code location in which they are thrown. Exceptions in the application layer, i.e. in system applications, cause the respective app to crash. Similarly as for normal apps, they can be restarted by the user. Sticky system app services are restarted automatically by the system. Exceptions thrown in bound services of the SystemServer are handled in a special way: A system watchdog thread com.android.server.Watchdog constantly monitors the core services like PowerManagerService and ActivityManagerService. In case of a crash or deadlock, it reboots the entire system. Uncaught runtime exceptions in unmonitored system services cause the Android Runtime, including Zygote and the SystemServer to be restarted (hot reboot) while the kernel keeps running—effects also described in recent research [75].

4.6.2.2 Scope of Protected Resources

To learn more about the scope of protected resources we analyzed how often a particular resource is used by different entry methods of the same class. For all protected resources of a concrete class, we aggregated the original entry points and compared this number to the total number of entry points for this class. Figure 4.4 shows the results for Android version 5.1 in which individual class numbers were summed up and grouped in 25% bins. About one-third of the protected resources are highly individual, as they are used by at most 25% of the entry methods of a class. However, at the same time, a large fraction (over 53% of all resources) were used by more than half of the class' entry methods. This implies that there is a high degree of shared code within sensitive entry methods and control-flows from different entry methods converge frequently.

The gained knowledge about protected resources is also an initial and crucial step towards future work regarding automatic security hook placement in the framework or verification of manually placed hooks in AOSP extensions such as ASM [71] or ASF [21]. These solutions do not only require a solid understanding of *what* is protected but also

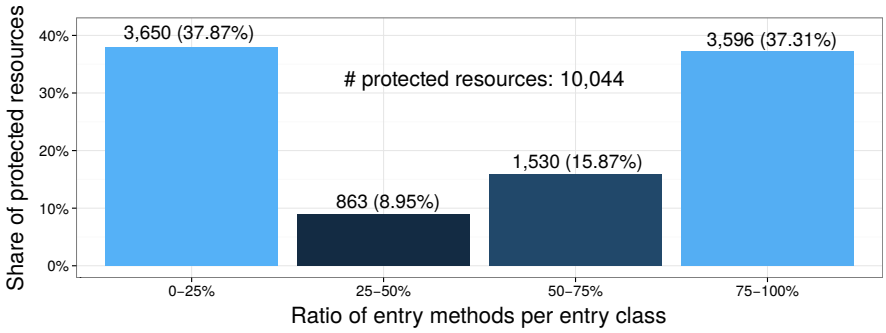


Figure 4.4: Usage of protected resources by ratio of entry methods per entry class for Android 5.1

where to place hooks and how to seed a minimal set of hooks while still completely mediating access to protected resources.

4.7 Permission Analysis

Building on top of our new insights we re-visit an important aspect of Android's permission specification, that is *permission mapping* between permission check and SDK method, and further, introduce *permission locality* to study which framework components perform which permission checks. To this end we extend AXPLORER as follows:

1) A PPEP only indicates the presence of a permission check in the control-flow from this entry-point, but there is no information yet about the number of checks or the concrete permission strings. We extend our slicing-based approach to also resolve the permission strings in common permission check API invocations (e.g., as defined in the Context class). Non-constant strings are resolved in a similar way like message codes in Section 4.5.2. From 520 distinct permission checks found in API level 16, we were able to resolve 99% of the permission strings. Among the failing cases, one case was located in the ActivityManagerService$PermissionController class where the permission string is an argument of the entry point method, which is only called from native code and hence was not statically resolved.

2) Entry class interconnection, i.e., RPC transitions to other PPEP (see Section 4.6.1.1), usually accumulates all permissions required by the additionally called entry classes for the UID that called the first entry class in the control-flow. However, those transitions are irrelevant for permission analysis when the RPC is located between calls to Binder.clearCallingIdentity and Binder.restoreCallingIdentity. Clearing the calling UID in the framework's bound services resets it from the calling app's UID to the privileged system server UID. Thus, outgoing IPC edges after clearing and before restoring the UID should be ignored in permission analysis, since the additional PPEP are called with a UID that is different from the calling app's UID.

3) We add a lightweight SDK analysis to reason about required permissions of documented APIs. To this end, we conduct a reachability analysis from public SDK methods to framework EPs (SDK to framework layer in Figure 2.1). Combining this mapping with the mapping from framework EPs to permissions creates a permission map for the documented API.

4.7.1 Re-Visiting Permission Mapping

The *Stowaway* project [113] was the first to generate a comprehensive permission map for Android 2.2. Their dynamic analysis approach (feedback directed API fuzzing) generates precise but incomplete results. Moreover, the involved manual effort makes it difficult to re-use it for newer API versions. *PScout* [20] improved on this situation by statically analyzing the framework code, thus increasing the code coverage. In direct comparison, *PScout*'s results contain notably more permission mappings. To handle the complexity induced by the framework size, *PScout* resorts to low-precision data models based on class hierarchy information. In the following, we demonstrate that this has negative implications for their resulting permission map. Using our insights we provide permission mappings that call the validity of prior mappings into question.

We compare our results with *PScout* using their latest available complete results (for Android 4.1.1). Since we exclude Intent and ContentProvider permissions, which both require supplemental analysis effort such as manifest or URI object parsing, we restrict the comparison to un-/documented APIs. For the evaluation we include the standard system apps and make identical assumptions as *PScout*, i.e., we assume that any permission found for a particular API is indeed required (a more precise analysis would require path-sensitivity). Moreover, like *PScout*, we did not conduct a native code analysis.

4.7.1.1 Documented API Map

Figure 4.5 shows for our documented API map (SDK EP to permissions) how often a certain permission is required. For some permissions, *PScout* reports higher numbers while for others AXPLORER reports higher numbers. Since the results are fairly deviating, we manually inspected various cases, including a full analysis of NFC and Bluetooth, to verify the correctness of our generated numbers. *PScout*'s higher method count, particularly for the two cases of NFC and Bluetooth, originates from adding package-protected methods that are not exposed to app developers and from improper handling of the @hide javadoc[2] attribute, resulting in an over-counting of the documented API methods. Our higher numbers for BROADCAST_STICKY and SET_WALLPAPER mainly refer to abstract methods from the Context class that are implemented in its subclass ContextWrapper and then inherited by 18 non-abstract subclasses (for API 16). Instead, *PScout* only lists those methods for the Context/-Wrapper class, thus missing to count the non-abstract subclasses.

[2]EP methods annotated with the @hide attribute are not included in the SDK.

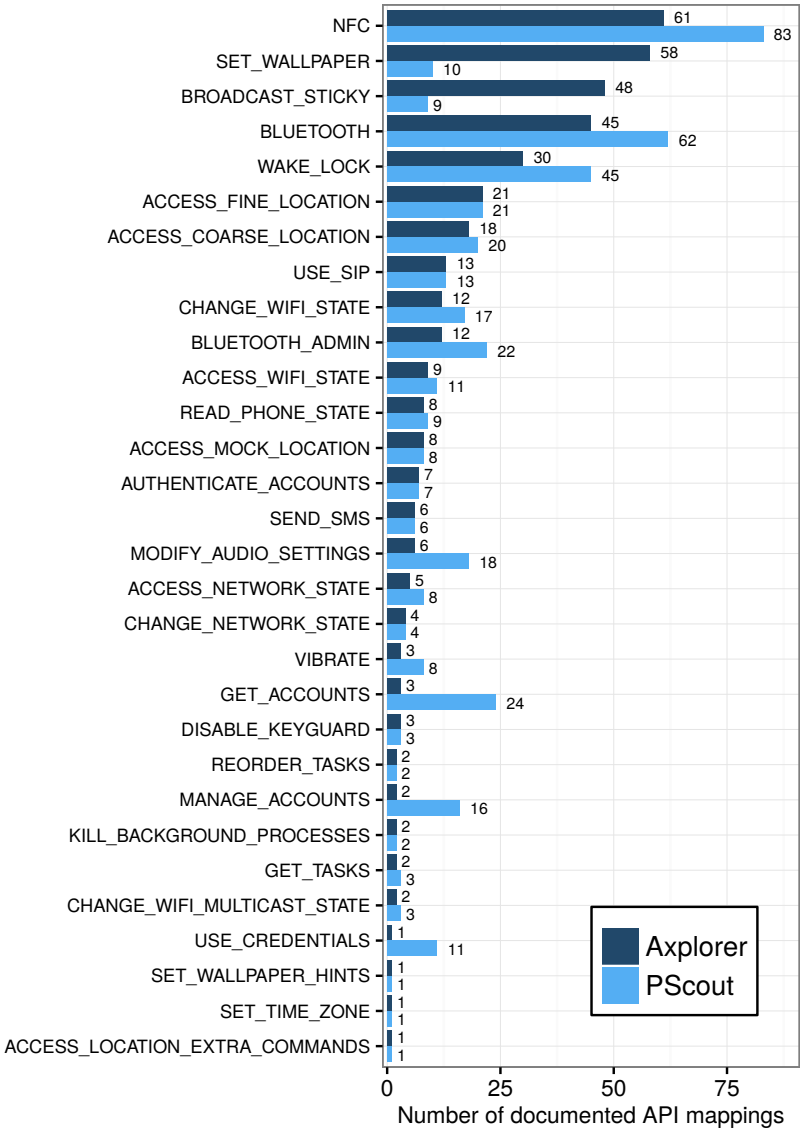


Figure 4.5: Number of documented APIs per permission.

Figure 4.6 provides a different view of the mapping by showing the distribution of required permissions per API. The main difference is the smaller number of outliers in our dataset: four mappings with three or more required permissions, compared to 58 such outliers in the *PScout* dataset. While the different results in Figure 4.5 mainly originate from technical shortcomings in the SDK analysis, Figure 4.6 hints at the different quality of the undocumented API map as result of a more precise framework analysis (see next Section 4.7.1.2). *PScout*'s more lightweight framework analysis results in an over-approximation of permission usage of EPs. For their outliers with more than five permissions in the ConnectivityManager class, they either over-approximate the receivers of a sendMessage call and/or did not resolve the message code and the

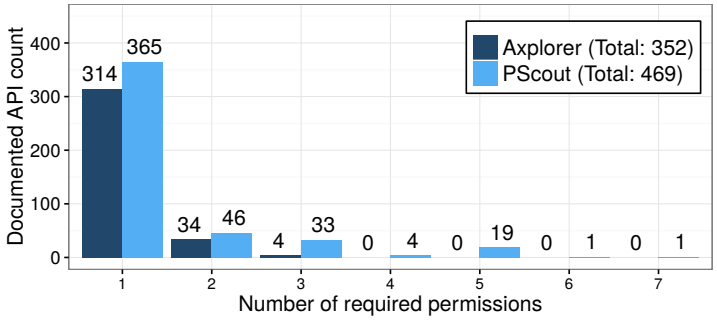


Figure 4.6: Number of permissions required by a documented API.

correct path in the handleMessage method. In such cases, the over-approximation in the framework analysis negatively influences the quality of the SDK map when IPC calls from the SDK to the application framework are connected. We manually validated all outliers and found that no method actually requires more than three permissions, thus contradicting the *PScout* results. Even the four outliers in our dataset check at most two permissions, independent of the EP call arguments. Additional permission checks might only be required for specific arguments/parameters. For instance, the setNetworkPreference(int) function of the ConnectivityService will tear down a specific type of network trackers depending on a preference integer argument. Some subtypes such as the BluetoothTetheringDataTracker require both Bluetooth permissions to execute this functionality while other subtypes require no additional permission. Adding parameter-sensitivity to the analysis is required to resolve such cases automatically and to annotate permission checks with conditions.

4.7.1.2 Undocumented API Map

A fair, direct comparison of permission maps for undocumented APIs is unfortunately very difficult due to shortcomings in the original paper. Although *PScout* did not explicitly define the term *undocumented API*, we assume, after manual inspection of their results, that it refers to the publicly exposed framework interfaces and covers any functionality that can be called from application level (independent of whether it is provided by SDK or system apps). Hence we refer to undocumented API as the entire set of framework entry points (cf. Section 4.4.1).

In contrast to *PScout*'s documented API map, we discovered different inconsistencies in their undocumented mappings. Besides valid mappings from PPEP to permissions, they also include mappings for unrelated methods. First, public methods of AIDL-based entry classes (which we define as *Entry Points*) are counted up to five times: once in the SDK manager class, in the framework service class, in the AIDL interface class, and in the auto-generated Stub and Proxy classes. Second, their mapping contains methods of StateMachine State classes. StateMachines are used framework-internally and their functionality is not exposed to apps. Third, synthetic accessor methods, as well as methods of anonymous inner classes, are reported. We assume that this problem is

related to the lack of a concise entry point definition that induces difficulties with the abort criteria during their backwards analysis starting from permission checks. In contrast, our forward analysis seems more suitable in this context, as permission checks are usually closely located to framework EPs.

Table 4.1 reports on the numbers of entry points per API level. For Android 4.1.1, Axplorer found 863 PPEP (33.4% of entry points) that require at least one permission. These numbers include signature/-OrSystem permissions since this information, although not interesting for app developers, is of interest for understanding the Android permission model in its entirety. On average we found 1.17 permissions per PPEP, which leads to a total of 1,012 permission mappings that cover 129 distinct permissions. This is a magnitude less than the 32,304 permission mappings reported by *PScout* for normal and dangerous permissions only. However, due to our more concise definition of what constitutes public framework functionality and the inclusion of all permission levels, we argue that our number is more substantiated.

4.7.2 Permission Locality

The application framework implements a separation of duty: every bound service is responsible for managing a certain system resource and enforcing permissions on access to them. For instance, the LocationService manages and protects location related information or the PhoneInterfaceManager facilitates and guards access to the radio interfaces. Permission strings already convey a meaning of the kind of system resource they protect and app developers might have an intuition for which services these permissions are required. To validate this aspect, we study whether permission checks follow the principle of separation of duty and whether permissions are checked by only one particular service. We call this aspect *permission locality*. A low permission locality indicates that a certain permission is enforced at different (possibly unrelated) services. This potentially contributes to the app developer's permission *in*comprehension that can lead to over-privileged apps [113]. Moreover, a strict separation of duty, i.e., high permission locality, significantly eases the task of implementing (and verifying) authorization hooks for resources, for instance in the design of recent security APIs [71, 21]. Consequently, the permission that protects a set of sensitive operations is ideally checked only in one associated entry class.

To study the permission locality, we analyze the checked permission strings and map them to the enclosing class of the permission check call. In Android v4.1.1 (API level 16) we found that out of 110 analyzed permissions 22 (20%) are checked in more than one class. Among these permissions, 13 are checked in two classes, 5 in three classes and 4 in four classes. An example for seemingly unrelated classes are LocationManagerService and PhoneInterfaceManager that both check the same dangerous permission ACCESS_FINE_LOCATION. While the permission is intuitively related to the first service, the connection to the latter one becomes obvious by looking at the enclosing method that includes the check (e.g. getCellLocation). Interestingly, PhoneInterfaceManager is not a framework service but included in the telephony system

app. Mixing framework services and system apps for enforcing identical permissions complicates permission validation and policy enforcement since system apps might be vendor-specific. Table 4.3 shows the number of permissions in API 16 that are checked in more than one class, grouped by their protection level. These numbers imply that there is no apparent correlation about the relative frequency and the protection level, in particular between the permissions available to third-party apps (normal+dangerous, 12/54) and the ones reserved for the system (10/56). This implies that low permission locality equally affects all protection levels. Applying this analysis to API 22 results in an

Protection level	**# permissions**	
normal	2/16	(12.5%)
dangerous	10/38	(26.3%)
signature	2/25	(8%)
signatureOrSystem	8/31	(25.8%)

Table 4.3: Number of permissions in API 16 that are checked in more than one class grouped by permission protection level.

even lower overall permission locality. Focusing on the four outliers in API 16, changes in API 22 include three class renamings, two removals and nine additions (cf. Figure 4.7). In case of READ_PHONE_STATE, those four classes even reside in four distinct packages of which one is part of the telephony system app. The permission CONNECTIVITY_INTERNAL more than doubled the number of classes (10) in which it is enforced. Besides renamed classes (blue color), as result of a code refactoring process, the number of additions for this small number of examples clearly indicates a disconcerting trend to lower permission locality. As a consequence, permission checks violate the separation of duty. This clearly emphasizes the need for centralized enforcement points.

A possible solution to increase permission locality includes associating each permission, in the ideal case, with a single service (though this might be challenging for all permissions). Once a designated owner service has been identified for each permission, a dedicated permission check function could be publicly exposed via its Binder interface, e.g., a method to check the ACESSS_FINE_LOCATION permission could be added to the ILocationManager interface. The addition and removal of callers to such methods then no longer affects the number of decision points and preserves the separation of duty for permission checks.

4.8 Discussion of Other Use-Cases

We briefly discuss further use-cases that can benefit from our work, particularly from our taxonomy of protected resources and the insights from our permission locality analysis.

```
Permission : ACCESS_NETWORK_STATE
Level      : normal
Checked in :
  - com.android.server.ConnectivityService
  - com.android.server.ethernet.EthernetServiceImpl
  - com.android.server.ThrottleService
  - com.android.server.net.NetworkPolicyManagerService
  - com.android.server.net.NetworkStatsService

Permission : READ_PHONE_STATE
Level      : dangerous
Checked in :
  - com.android.internal.telephony.PhoneSubInfoProxy
  - com.android.internal.telephony.SubscriptionController
  - com.android.phone.PhoneInterfaceManager
  - com.android.server.TelephonyRegistry
  - com.android.server.net.NetworkPolicyManagerService

Permission : CONNECTIVITY_INTERNAL
Level      : signatureOrSystem
Checked in :
  - com.android.bluetooth.pan.PanService$BluetoothPanBinderl
  - com.android.server.ConnectivityService
  - com.android.server.NetworkManagementService
  - com.android.server.NsdService
  - com.android.server.connectivity.Tetheringl
  - com.android.server.ethernet.EthernetServiceImpl
  - com.android.server.net.NetworkPolicyManagerService
  - com.android.server.net.NetworkStatsService
  - com.android.server.wifi.WifiServiceImpll
  - com.android.server.wifi.p2p.WifiP2pServiceImpll

Permission : UPDATE_DEVICE_STATS
Level      : signatureOrSystem
Checked in :
  - com.android.server.LocationManagerService
  - com.android.server.am.BatteryStatsService
  - com.android.server.am.UsageStatsService
  - com.android.server.power.PowerManagerService$BinderService
  - com.android.server.wifi.WifiServiceImpl
```

Figure 4.7: Permissions checked in four distinct classes in API 16. Colors denote changes in API 22: renamed classes (blue), additions (green) and removals (red).

4.8.1 Permission Check Inconsistencies

Prior work *Kratos* has shown that the default permission checks are inconsistent and can lead to attacks [120]. However, this approach explicitly did not make the attempt to identify protected resources in Android's application framework but instead relied on arbitrary shared code as a heuristic to identify security relevant hotspots in the framework's code base. While this approach has successfully demonstrated the need for such analysis, we argue that using our definition of protected resources as a refinement of shared code can further improve the precision of their analysis, since, by definition,

protected resources describe sensitive operations. False positives originating from shared logging or library code are automatically eliminated. Distilling a more concise definition of field-update and native method call resources from our high-level taxonomy is a promising future work. General heuristics such as the presented removal of non-relevant field updates of a this reference within constructors and the special treatment of instructions in the Java namespace could be replaced with machine-learning techniques to improve the highly individual classification task.

4.8.2 Authorization Hook Placement

Different Android framework extensions [103, 162, 107, 30, 71, 21] augment the application framework with authorization hooks in a best effort approach. On commodity systems, a comparable situation for the Linux and BSD kernels has been improved through a long process that established a deeper understanding of the internal control and data flows of those kernels and that allowed development of tools to verify or automate placement of authorization hooks. A similar evolution for Android's application framework has yet been precluded due to open technical challenges: first, one must be able to analyze control and data flows in the framework across process and service boundaries; second, one must be able to track the execution state of the framework service along its internal control and data flows (e.g., tracking the availability of the subject identity); third, one has to establish a clear and very specific understanding of the protected resources of each service. This work at hand addresses the first of these challenges and provides necessary permission locality information to implement comprehensive, coarse-grained enforcement models. Additionally, with our high-level taxonomy of protected resources, we made a first step towards solving the third challenge.

4.9 Related Work

Static (App) Analysis. Different related works have analyzed Android apps for vulnerabilities and privacy violations. Enabling precise static app analysis required solving essential questions like what are the entry points of the app, what are the security relevant sinks and how can we achieve a static runtime model that takes the application peculiarities into account? Among the static analysis approaches, *CHEX* [94] was the first tool to accommodate for Android's event-driven app lifecycle with an arbitrary number of entry points. *FlowDroid* [19] further improved the runtime model by automatically generating per-component lifecycle models that take into account the partial entry point ordering. While *FlowDroid* still analyzed components in isolation, a number of related works specifically addressed the problem of inter-component communication (ICC). The initial work *Epicc* [105] devised a new analysis technique to create specifications for each ICC sink and source. *Amandroid* [144] combined a lifecycle-aware program dependence graph with ICC analysis to generate an inter-component model of the application to improve precision for various security applications. Similarly, *IccTA* [84] extended *FlowDroid* with a precise inter-component model. Finally, *IC3* [104]

uses composite constant propagation to improve retargeting of ICC-related parameters enabling a more precise ICC resolution. Moving from best effort approaches, *SuSi* [18] took a machine-learning approach for classifying and categorizing sources and sinks in the framework code that are relevant for application analysis. All of those solutions contribute to analyzing Android apps more efficiently. The focus of this work is on establishing similar knowledge on Android's application framework and on making a first essential but non-trivial step towards enabling a holistic analysis of Android that includes the framework code with its security architecture.

Application Framework Abstractions. The application framework is generally regarded as too complex to be considered in an app analysis (cf., *CHEX* [94]) and very recent works dealt specifically with this problem of abstracting the application framework [31, 63] or making it amenable for app analysis [27]. *EdgeMiner* [31] links callback methods to their registration methods and generates API summaries that describe implicit control flow transitions through the framework. *DroidSafe* [63] distills a compact, data-dependency-aware model of the Android app API and runtime from the original framework code. *Droidel* [27] differs in its approach by explicating the reflective bridge between the application framework and applications, while trying to model the framework as less as possible. It generates app-specific versions of the application framework and replaces reflective calls with app-specific stubs. All of these approaches try to pre-compute data-dependencies through the framework API that can be used by app analyses in favor of using the complex and huge framework code base. In contrast, our work makes a first step towards enabling in-depth analyses of the application framework beyond just data dependencies in order to enable future reasoning about framework security architectures or extensions (such as guiding and verifying hook placement or separation of duties).

Permission Mapping and Inconsistencies. Both *Stowaway* [113] and *PScout* [20] built permission maps for the framework API. Stowaway used unit testing and feedback directed API fuzzing of the framework API to observe the required permission(s) for each API call. PScout, in contrast, used static reachability analysis between permission checks and API calls to create a permission mapping of different Android framework versions that improves on the results of Stowaway. Permission maps have since been a valuable input to different Android security research, such as permission analysis [65] and compartmentalization of third-party code [111, 121], studying app developer behavior [113, 139], detecting component hijacking [94], IRM [79, 23] and app virtualization [22], or risk assessment [109, 64, 156, 149]. In this work, we re-visit the challenge of creating a permission map for Android. In contrast to prior work, we build on top of our new insights on how to statically analyze the application framework (see Sections 4.4 and 4.5), which allow us to achieve a map that is more precise for the application framework API and that calls the validity of some prior results [20] into question. We discuss how recent work [120] that focused on inconsistent security enforcement within the framework could benefit from a deeper understanding of the framework's peculiarities separately in Section 4.8.

Android Security Frameworks. Various security extensions have been proposed, such as [103, 162, 107, 30, 71, 21] to name a few, which integrate authorization hooks into Android's application framework to enforce a broad range of security policies. At the moment, those extensions are designed and implemented as best-effort approaches that raise questions about the completeness and consistency of the enforcement and indeed past research has shown that even the best efforts of highly experienced researchers and developers working in this environment introduce potentially exploitable errors [51, 155, 125, 120]. This unsatisfying situation has strong parallels to earlier work on integrating authorization hooks into the Linux and BSD kernels [148, 143], where a dedicated line of work [51, 155, 57] has established tools and techniques to reason about the security properties of proposed extensions or to automate the hook placement. Prerequisite for those solutions was a clear understanding of what constitutes a resource that is (or should be) protected by an authorization hook. To allow development of similar tools for the Android application framework, we hence have to also answer the question about Android's protected resources first. In this work, we make a first essential step in this direction by enabling a deeper analysis of the framework and by providing a first high-level taxonomy of protected resources in the application framework.

4.10 Conclusion

In this work, we studied the internals of the Android application framework, in particular challenges and solutions for static analysis of the framework, and provided a first high-level classification of its protected resources. We applied our gained insights to improve on prior results of Android permission mappings, which are a valuable input to different Android security research branches, and to introduce permission locality as a new aspect of the permission specification. Our results showed that Android permission checks violate the principle of separation of duty, which might motivate a more consolidated design for permission checking in the future. To allow app developers and independent research to benefit from our results, we published our data sets at `https://github.com/reddr/axplorer` .

5

LibScout

Reliable Third-Party Library Detection in Android

5.1 Motivation

Third-party libraries on Android have been shown to be security and privacy hazards by adding security vulnerabilities to their host apps or by misusing inherited access rights to leak sensitive data. Correctly attributing improper app behavior such as privacy leaks (cf. Chapter 3) either to app or library developer code or isolating library code from their host apps would be highly desirable to mitigate these problems, but is impeded by the absence of a third-party library detection that is effective and reliable in spite of obfuscated code. This work proposes a library detection technique that is resilient against common code obfuscations and that is capable of pinpointing the exact library version used in apps. Libraries are detected with profiles from a comprehensive library database that we generated from the original library SDKs. We apply our technique to the top apps on Google Play and their complete histories to conduct a longitudinal study of library usage and evolution in apps. Our results particularly show that app developers only slowly adopt new library versions, exposing their end-users to large windows of vulnerability. For instance, we discovered that two long-known security vulnerabilities in popular libs are still present in the current top apps. Moreover, we find that misuse of cryptographic APIs in advertising libs, which increases the host apps' attack surface, affects 296 top apps with a cumulative install base of 3.7bn devices according to Play. This is the first work to quantify the security impact of third-party libs on the Android ecosystem.

5.2 Problem Description

Third-party libraries have become a fixed part of mobile apps. Developers use them to monetize their apps through advertisements, integrate their apps with online social media, include single-sign-on services, or simply leverage utility and convenience libraries for their apps' functionality.

However, third-party libraries are a double-edged sword: While they can provide convenience for the app developer and can greatly enhance their host apps' features, they also have been shown to be a hazard to the end-users' privacy and security. A number of prior studies [53, 65, 29, 118, 128] has demonstrated that such libraries exhibit questionable privacy practices. For instance, they leak user-private information, exploit their host app's privileges, or track users. Two recent incidents of such questionable practices were revealed in the popular SDKs of Taomike—China's biggest mobile ad provider—and Baidu, that were found to be secretly spying on users and uploading their SMS to remote servers [134] and opening backdoors to the users' devices [132], respectively. In addition to such privacy violations, third-party libs increase the attack surface of their host apps when they do not adhere to security best practices and, hence, become a liability for the users' security. In the recent past, even popular libraries by reputable software companies, such as Facebook and Dropbox, were affected by highly severe vulnerabilities. The found vulnerabilities could lead to the leakage of sensitive data to publicly readable data-sinks [110], code injection attacks [112, 62],

account hijacking [133], or linking a victim's device to an attacker-controlled Dropbox account [49].

Given the high prevalence of third-party libraries in apps and consequently their high impact on the health of the entire smartphone ecosystem, it is of no surprise that dedicated research has investigated new mechanisms to sandbox or remove libs, with a strong focus on advertisement libs [121, 111, 150, 118]. Yet, these proposals make one crucial assumption that currently limits them in their effectiveness: they assume that libraries can be clearly identified, either through developer input [111] or inspection of the app's code [121, 150, 118]. *Reliably* identifying libraries, however, forms a formidable and yet unsolved technical challenge. First, third-party libraries are tightly integrated into their host app by statically linking them during the app's build process into the app's bytecode, thus blurring the boundaries between app and library code. Second, app developers commonly make use of bytecode obfuscation tools, such as ProGuard [68]. One side-effect-free bytecode obfuscation technique is identifier renaming. It turns identifiers into short, non-meaningful strings, i.e. a package name *com.google* is transformed into *a.c.* Naïve library detection approaches based on identifier matching, like in [128, 65, 121, 150] or applied by third-party ad detector apps, fail even to this simple obfuscation technique.

Another problem, caused by the inability to reliably detect third-party libraries within applications, is the lack of accountability for privacy and security violations. For instance, a wide range of security-related analyses studied apps for privacy and security issues and raised awareness for various problem areas, including privacy leaks [59, 19, 144, 63, P1], permission usage [146], dynamic code loading [112], SSL/TLS (in-)security [106, 54], or (mis-)use of cryptographic APIs [52]. However, without being able to distinguish app developer code from third-party library code, the reported results are on a *per-app* basis and do not distinguish whether bad or improper behavior originates from app or library developers.

To increase the efficiency of library sandboxing mechanisms and to be able to hold the correct principal (app or lib developer) accountable for security and privacy violations, a reliable and precise third-party library detection is required that is resilient against common obfuscation techniques.

5.3 Contribution

In this work, we make two tangible contributions: First, we present an efficient and reliable approach for detecting third-party libraries within Android apps (see Section 5.4). By analyzing the original library SDKs, we extract profiles that are resilient to common obfuscation techniques, such as identifier renaming and API hiding. To achieve these properties our approach is based on class hierarchy information only and is independent of the libraries' code. Still, our profiles are fine-grained enough to not only detect distinct libraries, but also the exact version used in an app. For the actual library detection, we devised a profile matching algorithm that reports whether an exact copy

of a given library version was matched. In the negative case, either because the correct version is missing in our database or dead-code elimination was applied to app code, a similarity score indicates the best matching profile for the library code in the app.

Second, we use our library detection technique in a longitudinal study of third-party libraries included in the top apps on Google Play (see Section 5.6). In this study, we are interested in finding answers to security- and privacy-related questions about libraries, such as *"How prevalent are third-party libraries in the top apps and how up-to-date are the library versions?"*, *"Do app developers update the libs included in their apps and how quickly do they update?"*, or *"How prevalent are vulnerabilities identified in prior research [112, 52] in libraries and how many apps are affected?"* To answer these questions, we first built a comprehensive repository of third-party libraries and applications (see Section 5.5). Our library set contains 164 libraries of different categories (Advertising, Cloud,..) and a total of 2,065 versions. We then collected and tracked the version histories for the top 50 apps of each category on Play between Sep 2015 and July 2016, accumulating to 96,995 packages from 3,590 apps. We complemented this database with meta-information, such as app and library release dates, which we collected from public sources or developer websites. Based on this dataset, we show that app developers commonly neglect updates of third-party libraries. By analyzing the time-to-fix of two recent security/privacy incidents of the Facebook and Dropbox SDKs [133, 49], we show that this developer negligence in updating libraries exposes end-users to large windows of opportunities for attacks (e.g., on average 190 days for 51 apps with a vulnerable Facebook SDK in our dataset). Lastly, we scan our library set for the presence of API misuse vulnerabilities [112, 52] that would expose the libs' host apps to cryptanalytic and code injection attacks and discover 18 vulnerable libs (61 versions), which together affect 296 apps with a cumulative install base of 3.7bn. Overall, our work constitutes the first longitudinal security study of third-party libraries in the Android ecosystem. We provide the first valuable insights into the security-impact of libraries and as such motivate future work on improving library updatability on Android. In summary, we make the following contributions:

(1) We are the first to devise a lightweight and effective approach (LibScout[1]) to detect third-party libraries in Android apps that is resilient to common obfuscation techniques and capable of pinpointing exact library versions.

(2) We created a large third-party library database including 164 distinct libraries with 2,065 versions, which we profile. We collected the version history of 3,590 top apps on Play for a total of 96,995 distinct packages. We complement those databases with meta-data such as app/library release dates.

(3) We conduct a longitudinal study of third-party libs in our app set to investigate their prevalence, the update frequency of apps and of libs, as well as the impact of app popularity and library API stability on the lib update frequency.

(4) We study time-to-fix and vulnerability windows of apps that include vulnerable

[1] https://github.com/reddr/LibScout

library versions (at the example of recently reported incidents of the popular Facebook and Dropbox SDK). Our results show large windows of opportunity for attackers against apps including those libraries.

(5) Lastly, we re-apply existing app analysis techniques to library code to investigate the improper usage of dynamic code loading and crypto APIs. We identify 61 library versions that affect 296 top apps on Play with a cumulative install-base of 3.7bn users and expose these apps to cryptanalytic attacks.

5.4 Technical Problem Description and Approach

In the following, we first aggregate requirements for a reliable and precise third-party library detection, before we describe our concrete approach in detail.

5.4.1 Requirements Analysis

Version Detection & Similarity Computation. Related work on library detection largely strove for completeness by reducing this problem to code similarity between apps. While this works well for detecting components within an app, such approaches always suffer from uncertainty as they are not based on the ground truth in form of the original library code. Moreover, this lack of ground truth precludes a more fine-grained detection including inference of the concrete library version. This information, however, is imperative to conduct longitudinal studies on the Android ecosystem to analyze library updatability, compatibility, and identifying vulnerable versions.

Therefore, we base our approach on the original library code provided as SDK by the library provider. Although relying on the original libraries comes with the drawback of incompleteness, particularly for less prevalent libraries, it is a necessary trade-off to infer concrete library versions, which would not be possible without ground truth. In addition and in contrast to common belief, libraries are often *not* included as is but in some modified form. In most of the cases, this refers to bytecode optimization through dead-code elimination of unused library functionality. To reliably detect such partially incomplete code fragments in apps, ground truth is required to compute a similarity value between library code in the app and original lib versions. To this end, we build a comprehensive library database (see Section 5.5) including the library binaries and complementary information, such as library name, version, and release date (if available).

Robustness Against Common Obfuscation Techniques. A widely-used third-party library detection technique is naïve package name matching. While package name matching provides a rough estimation of the different components within an app, this approach comes with several drawbacks. In the presence of code obfuscation, such as the commonly used identifier renaming, detection fails since the original package

	Allatori [122]	**dashO** [114]	**DexGuard** [67]	**DexProtector** [88]	**DIVILAR** [159]	**ProGuard** [68]	**Stringer** [89]
API hiding (*)	✘	✘	✓	✓	✘	✘	✘
Class encryption	✘	✘	✓	✓	✘	✘	✘
CF randomization (*)	✓	✓	✘	✘	✘	✘	✘
Identifier renaming (*)	✓	✓	✓	✓	✘	✓	✓
String encryption (*)	✓	✓	✓	✓	✘	✘	✓
Virtualization	✘	✘	✘	✘	✓	✘	✘

Table 5.1: Feature comparison of Android app obfuscators. Our approach is robust against features marked with (*).

names no longer exist. Some libraries, such as the advertising lib *airpush*, even use a similar technique to defeat such naïve detection methods. To get an overview of available obfuscation techniques for Android other than identifier renaming, we studied popular obfuscation tools and their capabilities. A high-level feature comparison is presented in Table 5.1 (features like application water-marking or resource file encryption are not relevant for our analysis and therefore omitted). The summary shows that code-based detection approaches additionally would have to cope with different kinds of code-obfuscation techniques, such as reflection-based API hiding or control-flow (CF) randomization. Ideally, a detection approach should be resilient to all presented obfuscation techniques. However, depending on the analysis technique (static or dynamic) inherent limitations apply.

Our approach is based on static analysis, hence dynamic code loading or class encryption to dynamically create code at runtime are general limitations. Similarly, virtualization-based protection, i.e., dex bytecode is replaced by a virtual instruction set that is interpreted by a custom virtual machine [159], is out of scope. To still handle the most common obfuscation techniques (marked with (*) in Table 5.1), our approach solely relies on class hierarchy information and does not depend on the actual library code. Since the integrity of library code (e.g., piggybacking malware) is not a concern of this work, we do not face drawbacks from the design decision to rely on class hierarchy information.

The next section gives detailed information on our lightweight detection approach that fulfills these requirements, while Section 5.6 evaluate our prototype in terms of accuracy, memory requirements, and performance.

5.4.2 Library Detection

The workflow of our approach consists of two separate steps. We first extract profiles from any original library version in our repository. Given this set of profiles, we statically detect libraries in apps by extracting app profiles and subsequently apply a matching algorithm to check whether and how libraries match. Profiles are generated from class hierarchy information only and do not rely on concrete library code. This is necessary to be robust against code-based obfuscation techniques such as control-flow randomization or API hiding. Our profile matching algorithm reports a *similarity score* between 0 and

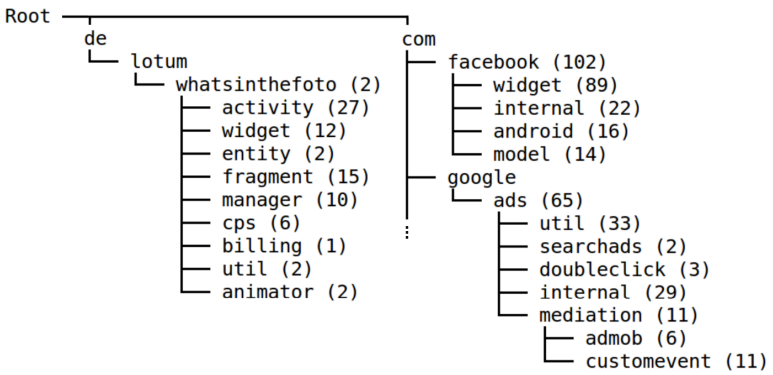


Figure 5.1: Generated partial (unobfuscated) package tree of the app de.lotum.whatsinthefoto. Numbers denote the classes per package.

1, indicating whether a library version was matched exactly, partially or not at all for a given application. The remainder of this section provides detailed information on the underlying data structures for the profile generation and the actual profile matching algorithm.

5.4.2.1 Package Tree

Java packages are a technique for organizing classes into namespaces. Packages are defined using a hierarchical naming pattern with levels in the hierarchy separated by dots. Packages that are lower in the hierarchy are usually referred to as subpackages. By convention, package names are written in lower case and companies should use their reversed Internet domain name as leading package, i.e. google.com uses a package name com.google.*. App and library developers usually stick to this convention or at least provide a namespace that is unlikely to appear in a different software component. This simplifies library integration in which lib code is statically linked during the app's build process.

The structure of the package hierarchy (often depicted as a tree) therefore gives a rough estimation on the included libraries in an application. Meta-information such as package relationships (parent, sibling, child/subpackage) and number of classes per package are not part of our profiles but will be used to improve the accuracy of our matching algorithm. Figure 5.1 shows a partial, un-obfuscated package tree of the app de.lotum.whatsinthefoto. It includes two third-party libraries in the com namespace—the Facebook SDK and Google Admob—and the app code in the de namespace.

We generate the package tree by performing a standard class hierarchy analysis (CHA). For each application class, we parse its package, i.e. for a class com.google.ads.AdView we receive an array of package fragments (com, google, ads). Starting from the root node, tree nodes are traversed in the order of the package fragment array. Non-existing child nodes are generated on-demand. Once the array is processed the class counter of the current tree node is incremented. This tree representation is used for debugging

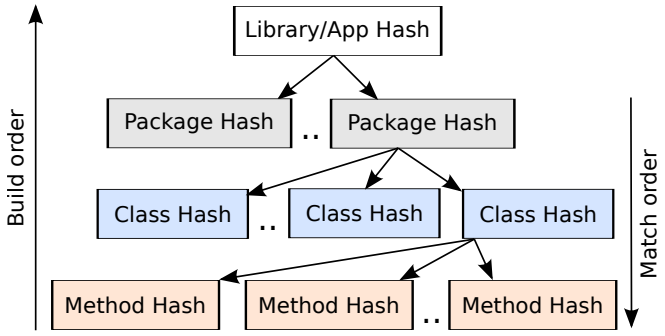


Figure 5.2: Merkle tree with a fixed depth of three. Tree is built bottom-up starting with method nodes. Matching is done top-down, depending on the preferred precision.

purposes and to perform structural checks during profile matching.

5.4.2.2 Profile Extraction

For the actual profiles we use a variant of Merkle trees [98]. In these hash trees, every non-leaf node is labeled with the hash of its child nodes. Our tree has a fixed depth of three where layers represent packages, classes, and methods (see Figure 5.2). In contrast to the package tree, our Merkle tree is flattened at the package layer, i.e. each distinct package is a child of the root node. This allows an efficient and precise package-based comparison between original libraries and applications to be tested.

In general, Merkle trees are used to verify contents of large data structures. To this end, the initial hashes in the leaf nodes are generated by hashing a piece of data, e.g. the content of a file. However, relying on actual code makes our approach susceptible to code-based obfuscation such as API hiding or control-flow randomization. Therefore, the method hashes have to be computed from non-obfuscatable information. For that, we use pruned method signatures. A signature is a string that uniquely identifies a method within an app. Figure 5.3 shows an example signature of method `open` in class `Session` of package `com.fb`. The method name is followed by the argument list in brackets and the return type. In a first step, we remove anything but the argument list and return type which is called *descriptor*, since any information before the argument list may be subject to identifier renaming. In the final step, we replace any non-framework type (framework types can be looked up in the Android SDK) by the same placeholder identifier `X` that can not serve as a type. This *fuzzy descriptor* keeps any non-obfuscatable information, but different application types are no longer distinguishable. The advantage is that we do not have to record a global type mapping but this also implies that a single fuzzy descriptor may match other methods as well. However, the introduced fuzziness for a single method is compensated by including all methods of a particular class. The higher the number of children (e.g. methods, classes) for a specific node, the smaller is the probability that two different nodes with the same number of children match.

To provide a deterministic hash generation for non-leaf nodes, child hashes are sorted

```
      signature: com.fb.Session.open(Activity, bool, com.fb.Session$Callback)com.fb.Session
     descriptor:                    (Activity, bool, com.fb.Session$Callback)com.fb.Session
fuzzy descriptor:                    (Activity, bool, X)X
```

Figure 5.3: Transforming a method signature into a fuzzy identifier that is robust against identifier renaming. The fuzzy descriptor constitutes the list of argument and return types in which any non-framework type is replaced by a placeholder **X**

and concatenated before being re-hashed. For the hashing, we found that the 128-bit MD5 hash algorithm provides a good trade-off between efficiency and precision for our use-case. As a default, we store the hash tree excluding the method layer to retrieve space-efficient profiles. If maximum precision is required, method hashes can be stored with their original signatures. In addition, we introduce a publicOnly mode in which only public methods of public classes are considered during tree generation. While the resulting profile is less unique, it is, at the same time, robust against changes of the internal API. We use such profiles to check library API compatibility in Section 5.6.

Bytecode Normalization Before building the hash tree of a component (library or app), we normalize its bytecode, in particular, we remove anything compiler-generated. This refers to bridge and synthetic methods such as accessor methods in nested classes with private attributes that are accessed by the enclosing class. This way we focus on developer-written code only and abstract from concrete compilers (usually javac for packaged libraries and dx for apps).

Some libraries have other library dependencies (e.g. *OkHttp* requires *Okio* for some I/O classes). In apps, those dependencies are resolved through static linking. If we analyze *OkHttp* in isolation, any type specified in *Okio* will not appear in the class hierarchy. For generating a fuzzy descriptor this is not a problem as we can treat such (non-existing) dependencies like normal library code. Hence, the application and library profile will not differ in this regard. This is not true when non-existing types are used as superclasses in library code. This may happen if the superclass is defined in a different library, e.g. the Fragment class in the *Android support v4* library. If a library L is analyzed in isolation, the class hierarchy builder ignores classes with unresolvable superclasses since the hierarchy can not be traced back to the root class java.lang.Object. However, when an app includes the *support v4* library in addition to L, the Fragment class is available and all its subclasses are added to the hierarchy. This results in a mismatch in the number of classes for the exact same library (version). To fix this problem without having to resolve dependencies during library profile extraction, we replace such non-existing superclasses with the root class. This allows the CHA to include such classes without causing side-effects in the generated profiles.

5.4.2.3 Profile Matching

Our matching algorithm tests whether and how a given library profile matches an app profile. To this end, for each library profile, a *similarity score* between zero and one

is computed, one indicating an exact library match. In contrast to profile extraction, matching is performed top-down in the Merkle tree. For testing whether a library *exactly* matches parts of the application profile, it suffices to check whether all library package hashes are included in the hash tree of the app. It becomes more complicated if an application only partially matches the library code, e.g. if the app includes a library version that is not in the library database or only parts of the original library are included (as result of a dead-code elimination). This implies that only a subset of package hashes matches and the similarity score drops below one. The higher the score is, the better a given library version is matched. If exact matching fails, the similarity score is computed at a deeper tree level, either on class or method level depending on the desired precision. We define the similarity score on class level between a library package lp and an app package ap as follows:

$$score_c(lp, ap) = \frac{\#\text{ classes in lp that match in ap}}{\#\text{ classes in lp}} \in [0, 1]$$

This definition tolerates the addition of classes, i.e. the score does not change if the application package has more classes than the library packages. However, if the app package contains fewer classes the similarity score will be smaller than one. Given that package hashes may no longer match, the question is which app packages should be compared to which library packages. A naïve approach would exhaustively compute the similarity score for any library/app package combination and take the global maximum, i.e. the set of best matching app packages. This might introduce false positives when matched app packages do not have the same root package and/or the package hierarchy is not preserved. For two library packages {com.google, com.google.ads} valid (obfuscated) candidate packages include {a.b, a.b.d} but neither {a.b, a.c} nor {a.b, c.d}. While a.b might be a valid candidate for com.google, the combination with a.c is invalid as the package hierarchy is no longer preserved. In the second invalid example, candidates do not have the same root package (a/c). To overcome this problem we apply the following four-step approach:

1) Candidate list. We first compute for each library package (lp) a list of candidate app packages (ap). An app package is a candidate if at least 50% of its class hashes match (configurable). An example could look like this:

$$\begin{aligned} lp_1\text{: } & ap_1(0.95),\ ap_2(0.84),\ ap_3(0.75) \\ lp_3\text{: } & ap_6(0.91),\ ap_4(0.60) \\ lp_2\text{: } & ap_7(0.85),\ ap_9(0.82) \end{aligned}$$

Every candidate list is sorted by score. In addition, library packages are sorted by similarity score of their highest candidate match.

2) Package linking. If a library package lp_1 with package name com.foo has app package candidates starting with the same package name, we can remove candidates with different root packages. In case identifier renaming has been applied to the app code, this direct linking no longer works. For filtering invalid combinations we therefore have to identify potential root packages. If lp_1 matches ap_1 with package name a.b.c we deduce that a.b is one potential library root package within the app. By applying

this to all pairs $<lp_i, ap_j>$ we receive a list of potential root packages.

3) Partitioning. Instead of exhaustively testing all combinations like in the naïve approach, it suffices to compute the maximum for each partition/root package and then take the global maximum. For each root package, we pre-filter the candidate list and remove any candidate that does not start with the current root package. For the remaining list, the maximum is computed exhaustively via backtracking. To eliminate combinations that do not structurally match the library package hierarchy, we define an abort criterion by testing structural equivalence between app packages and library packages. More formally, backtracking is aborted if the following requirement is violated:

$$\begin{aligned}&\forall ap_i, ap_j, lp_x, lp_y,\\ &ap_i \ candidateOf \ lp_x, \ ap_j \ candidateOf \ lp_y,\\ &relationship(ap_i, ap_j) = relationship(lp_x, lp_y)\end{aligned}$$

$relationship(p_1, p_2)$ tests whether p_1 is parent, sibling, or child of p_2 and in case of a parent/child relationship it further determines the package distance (an immediate subpackage has distance 1). If, for example, the backtracking algorithm traverses ap_1, ap_6 the calculation is aborted if $relationship(ap_1, ap_6) \neq relationship(lp_1, lp_3)$.

4) Global maximum. Finally, we select the maximum score over the partitions and sum up the matched classes (denoted as sum-classes). The *similarity score* on class level is consequently computed as

$$simScore_c = \frac{\text{sum-classes}}{\#\text{ of classes in library}} \in [0, 1]$$

We classify a library as partially matched if the score exceeds a minimum threshold such as 0.6. This value was determined experimentally to find a good trade-off between false-positives and false-negatives since a low similarity score might either result from dead-code removal (partial library inclusion) or from a library version detected that is not in the database. To increase precision, the similarity score can also be computed on method level if method hashes are included in the profiles.

5.5 Library and App Repository

In this section, we explain how we established a database of third-party libraries and applications which we subsequently use to evaluate our tool LibScout (Section 5.6) and on which we conducted a longitudinal study of third-party libraries used in the top apps of Google Play (Section 5.7).

5.5.1 Library Database

The foundation of our approach is detecting known libraries in Android apps. This requires setting up a library database that contains the ground truth in the form of original code packages for each available library version.

Identification and Retrieval of Popular Libs The first task to build a library database is to identify popular libraries, such as libs that are specifically developed for Android (e.g., ad and analytics libs) or Java support/utility libraries. In particular, advertising libraries are one of the most prevalent library types for Android. Google's developer documentation on *AdMob* mediation networks[2] gives a good but incomplete view on available ad libraries that are compatible with its own *AdMob* library. Another major source of library statistics is provided by the Android third-party market *AppBrain*[3]. It provides an *Ad Detector* app to gather statistics about the market share of popular libs. Libraries are identified by checking for well-known identifiers, such as package names, and are categorized into the three groups (ad networks, social SDKs, and development tools). In addition to such readily-available information, we manually analyzed the package trees (cf. Figure 5.1) of 50 popular apps to identify additional libraries based on package names.

Following this first bootstrapping step, we retrieve the identified library binaries and, if possible, their complete history, since our approach relies on the ground truth in the form of the original library code. We found that there are different ways how library developers distribute their SDK. More and more libraries can be found on the *Maven Central* repository or are hosted on public *GitHub* repositories. In these cases, it is trivial to retrieve the entire history. Other libraries, such as the *Facebook* SDK, are hosted directly at the library provider's website and might have migrated to *Maven* (as is the case for the *Facebook* SDK). Early versions of some libraries were distributed as open-source only (without pre-compiled binaries), hence it took some effort to compile each version with varying build environments. It gets more complicated if developer accounts are required to download a specific library (this is common for advertising libraries like *Tapjoy* or *Flurry*). Moreover, only the most current versions of some libraries were available. To still retrieve older versions, tricks like URL modification, searching for lib versions in known Android projects, or using the *Web Archive*[4] to access older versions of the download pages were required.

For each library version, we store the binary code, name and auxiliary information like version number and release date, which are usually available via change log or directly from the host server. Moreover, we categorize each library by functionality: Advertising, Analytics, Android, Cloud, Social-Media, and Utilities. The Android group contains support and Play-service libraries as well as libraries with custom UI widgets.

Library Statistics Our database contains 164 distinct libraries with 2,065 versions. Table 5.2 shows the distribution of library/-versions across categories. The database includes the most popular libraries for each category. For about 26% of all libraries, we got less than four versions, however, at the same time the database contains more than seven versions for 55% of the libraries. The mean number of versions per library is 12.59±1.09. For the advertising library *Heyzap* we were able to collect 96 distinct

[2] developers.google.com/admob/android/mediation-networks
[3] http://www.appbrain.com/stats/libraries/
[4] https://web.archive.org/

Category	# Libraries	# Library Versions
Android	51	560
Utilities	41	746
Advertising	40	337
Cloud	16	149
SocialMedia	9	149
Analytics	7	124
Total	164	2,065

Table 5.2: Number of distinct libs and versions per category

versions. For 2,026 (98.11%) of the library versions in our database, we were able to collect their release dates. From those release dates, we derive that the developers of the third-party libraries in our dataset release a new version on average every 117 ± 60 days and just in the first half of 2016 on average every 77 ± 20 days.

On a commodity laptop, the average time for profile extraction is 2.8 seconds per lib version. The mean number of packages, classes, and methods in our library set is <13, 304, 1,701>. This even exceeds the code base of many smaller apps. Outliers include the Google Play Service library with <85, 3,416, 18,794>. These results indicate that many libraries are very complex and/or offer a lot of functionality.

5.5.2 Play Store Crawler and Repository

Next, we provide an overview on how we selected our sample applications from the Google Play Store and built a longitudinal version history of these apps.

Upper Bound for Version Code We bootstrapped building our set of sample applications and their history by crawling the top 50 apps for each of the 20 categories on the Google Play Store in 2-hour intervals between September 2015 and July 2016, resulting in 4,666 distinct apps, which we continue to track even after they left the top 50 lists. We opted for this approach because prior studies [140, 158] have established that the Google Play Store is a "superstar" market in which a small percentage of the free applications (i.e., the top apps) account for almost all of the downloads. Thus, our sample set represents the apps with the largest user bases on Play—together accounting for almost 46 bn downloads by July 2016 according to Play.

To build the version history for every discovered application, the Google Play API can be iteratively queried for lower version codes. For instance, when our initial app set contained the application package *org.wikipedia* with version code *10*, we can iteratively request to download versions *9, 8, 7, etc.* of this app package from the server. Unfortunately, the versioning rules[5] for Android apps do not require app developers to follow any specific scheme except that the version code must be monotonically

[5] developer.android.com/tools/publishing/versioning.html

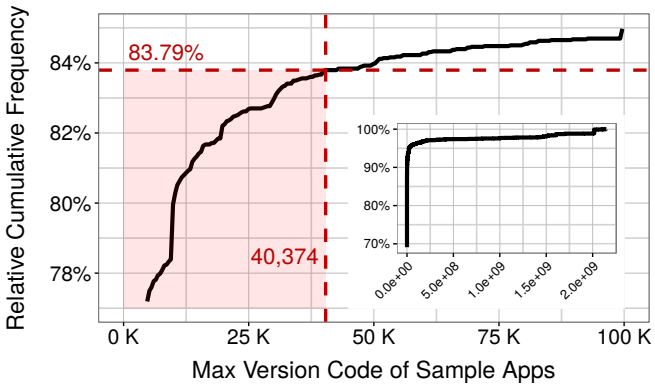


Figure 5.4: Distribution of maximum version codes in our initial app set and selected threshold for our crawler.

increasing between updates. This implies that version codes can be distributed all over the integer value range. As a consequence, we have to determine a reasonable upper bound for the version code to achieve a good trade-off between coverage of package version histories and the required time to build the history. Figure 5.4 illustrates the relative cumulative frequency distribution of maximum version codes in our app set. While most developers choose their version codes from the lower end of the possible value range, some developers choose version codes from within the range of millions to billions, accounting for the long tail of version codes (see the subplot in Figure 5.4).

For our study, we decided to create histories for apps at the lower end with a maximum version code of 40,374, providing a coverage of 3,910 apps (83.79%) of all apps in our set. Moreover, it is noticeable, that the long tail of version codes has a jump in the CFD around version code $2 * 10^9$. This stems from the fact that the developers of 64 apps in our set chose version codes based on the release date (e.g., following the pattern *YYYYMMDDVV*, where *VV* is the revision-per-day). We additionally included those 64 apps from the long tail of version codes, since their code immediately reveals the app version's release date and we built their history by iterating version codes in date-format from the discovered version back to Jan 1, 2012. Those apps increase the coverage by 1.37%, for a total coverage of 85.16%.

Sampled Version Codes

Figure 5.5 provides an overview of our application sample set after downloading all available versions for the top apps in the initial set. Overall, we have 96,995 packages for 3,590 distinct apps (excluding apps that have only one version available in our set). This results in an average of about 27 versions per app with a maximum of 754 available versions of the app *com.imo.android.imoimbeta*.

Release Dates and Update Frequency

As a last step in building the sample set for our longitudinal study of apps, we complemented our sample app database with release dates for each app version. Since Google Play only provides the release date for the most recent version of an app, we collected the release dates of older app versions

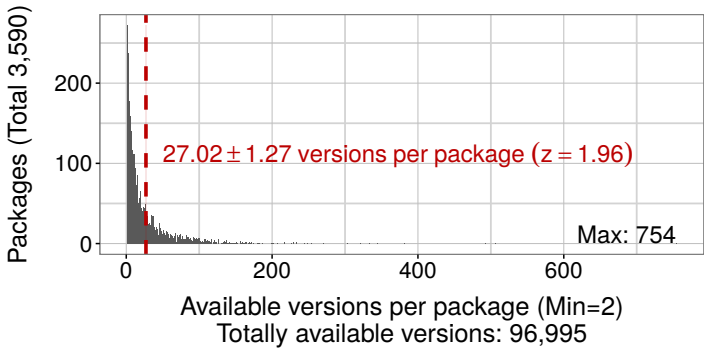


Figure 5.5: Sampled version codes and version codes per app.

from market analysis services such as `appannie.com`, `apk4fun.com`, and `appbrain.com`. In total, our database contains the release dates of 75,339 distinct packages (77.67% of 96,995 packages) for the 3,590 apps for which we retrieved older versions, where the release dates range from 12/19/2009 to 07/29/2016.

Based on those release dates, we estimate that the developers of the apps in our sample set release an app update, on average, every 62±2.94 days, where the average update frequency per app has increased since 2010 (e.g., 38±1.53 days in 2015 and only 29±0.99 days in first half of 2016).

5.6 Evaluation of Library Detection

We implement our approach on top of the WALA framework[77]. Our tool LibScout requires an additional 3.5 kLOC.

5.6.1 Library Profile Uniqueness

We start by answering the question on how effective our profiles are for distinguishing different library versions and evaluate the memory requirements for storing our profiles.

Since our profiles are generated with class hierarchy information only, it is possible that different library versions have the same profile when only the code has changed but no method interfaces of the public or private APIs of the lib. In these cases we still detect the library but report the set of possible versions. For 53/164 (32.3%) of libraries, all versions have unique profiles. For the 2,065 library versions in our repository, we found that 1,225/2,065 (59.3%) of profiles are unique, i.e., we can unambiguously pinpoint the exact library version. About 40% of all profiles are ambiguous, i.e., there exist at least two versions with the same profile. All ambiguous versions occurred in clusters of consecutive versions. Such clusters are expected for version updates in which only bugfixes and minor code changes are implemented. The average size of such clusters is 2.77 versions and only two exceptional cases (*Amazon Analytics* and *braintree payments*) exist with a cluster size of 10 in each case. Although we cannot pinpoint the exact library

#APKs	Ratio	Library	Category
41,518	41.22%	Facebook	Social media
30,310	30.10%	Gson	Utilities
18,026	17.90%	Flurry Analytics	Analytics
16,336	16.22%	Bolts	Utilities
14,229	14.13%	Crashlytics	Android
14,146	14.05%	OkHttp	Utilities
13,758	13.66%	Nine Old Androids	Android
12,201	12.11%	Facebook Audience	Advertising
11,066	10.99%	Picasso	Android
9,694	9.63%	Retrofit	Utilities

Table 5.3: Top 10 detected libraries in our app repository, excluding Google support and play service libs.

version for ambiguous profiles, we significantly reduce the search space for post-analyses (e.g., code inspection) to 2–3 candidate versions on average.

The size requirement for storing a single profile is linear to the number of packages, classes, and methods of a library/app and additionally depends on the chosen hash function (e.g. 128 bit MD5). During our experiments, the precision of our approach did not change with larger hashes. The largest profile was generated for the *Adrally* Ad SDK (v2.2.0) with a size of 220 KB (normal profile without methods). Including methods and full debug info, such as the original method signatures, increases the size to a total of 3.1 MB. However, the average normal profile size is only ≈22 KB.

5.6.2 Library Prevalence

We apply LibScout with partial matching on our app dataset to study the prevalence of libraries. In addition, we automatically extract the root package for each library and apply a naïve package name detection for cases in which our profile matching does not report a result. Moreover, we use the package matching to validate the detection rate of the profile matching. Table 5.3 shows the top 10 detected libraries of our repository in absolute and relative numbers. We excluded Google support and Play service libraries as they represent eight of the top 10 libs, with the Android support v4 library leading with about 80%. Moreover, six Play service libraries would have been listed in the top 10, since many developers typically include the complete set of these libraries although not making use of it. The new list is led by the popular *Facebook* SDK that is included by about 40% of all apps. Further, the list includes advertisement (*Facebook Audience*) and tracking libraries as well as commonly used utility libraries such as *Gson*, *Bolts*, and *OkHttp*.

On average, we detect 13.1 distinct libraries/app. The app com.science.wishboneapp (various versions) contained the highest number of libraries (55). By reporting the numbers for profile-only detection we receive an average of 9.7 libs/app while the naïve approach finds an additional 3.4 libs/app. There are two main reasons for our approach

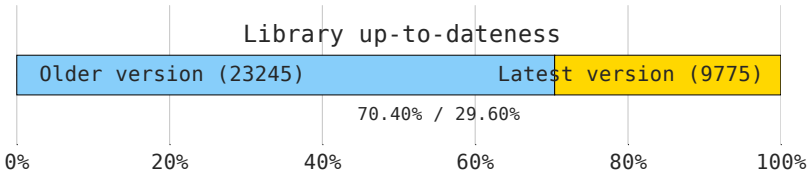


Figure 5.6: Up-to-dateness of included lib versions across the most recent versions of all apps in our repository.

to not detect a library via profile matching. The first reason is an incomplete set of library versions in our database. This particularly applies to advertising libraries that are not publicly retrievable, i.e. it is generally difficult to build a complete library history. The second reason is that code optimization has been applied during an app's build process to remove unused library code. If the app contains less than 60% of the original lib code our partial matching algorithm no longer reports a match. Finally, we checked whether the number of libraries per app evolves over time. To this end, we determined the average number of libraries on the set of earliest app releases and most current app releases. Between these two sets, LibScout reports a slight increase of 3.4 on the average library count, i.e. there is a trend to include more libraries.

5.7 Study of Third-Party Libraries

Lastly, we conduct a longitudinal study of third-party libraries in our app set to investigate important security-relevant questions such as *"How up-to-date are libraries used in top apps?"*, *"How quickly do app developers react to discovered vulnerabilities in their included libraries?"*, and *"How prevalent is the misuse of security APIs in third-party libs?"*.

5.7.1 Up-to-dateness of Libraries in Top Apps

For the most recent versions of all apps in our dataset, we checked whether included third-party libs are up-to-date or outdated with respect to the app release date. Figure 5.6 summarizes our results. In almost three-quarter of the cases (23,245 lib inclusions or 70.40%) the app developer included an outdated lib version, where the delta to the most recent library version is one in only 7.99% of all cases and in the most extreme case 81 versions. In terms of time difference between library version release and adaption by apps, the apps in our dataset required 324 ± 1 days on average to include a new library version. This is a rather poor adaption of newly released lib versions, particularly when considering that app developers release new versions, on average, almost twice as frequent as lib developers (see Section 5.5). Thus, we were interested in what potential factors could influence the adaption of a library. In particular, we investigated the library distribution channel, app popularity, and the libraries' public API compatibility.

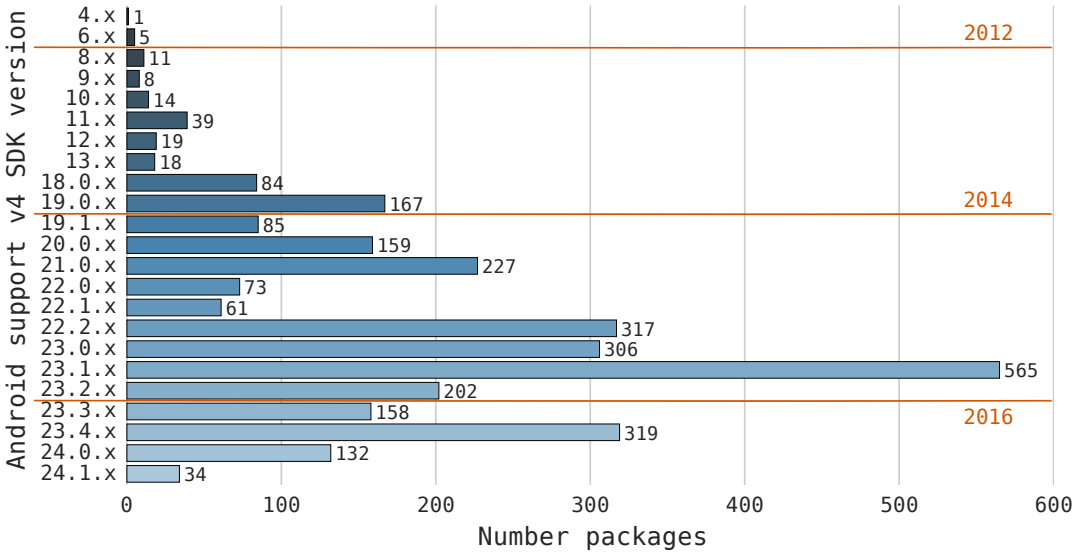


Figure 5.7: Distribution of Android support v4 SDK versions for the current top apps on Play. Orange bars indicate the beginning of the labeled year.

Distribution Channel and App Popularity A first potential factor is the distribution channel of the library (cf. Section 5.5). Two popular libraries, Android support v4 and Facebook (see Table 5.3), are employing opposing strategies: Android support v4 is shipped with Android IDEs (e.g., Android Studio and ADT) and is automatically added to apps based on their supported Android API levels, while Facebook prior to version 3.23.0 had to be manually downloaded from the developer pages and can since version 3.23.0 be retrieved via the Maven central repository. For Android support v4, we found that there is a clear bias towards newer library versions among the top apps (see Figure 5.7), while the majority of the top apps contain a (highly) outdated version of the Facebook SDK (see Figure 5.8). This indicates that app developers attend more carefully to the up-to-dateness of their IDE and its shipped packages than to manually retrieving external libraries (even from a central repository).

As a potential second factor, we considered the app popularity measured in the app's number of downloads. However, we could not discover any effect (Kendall's $\tau = 0.01$ with $p = 0.8 \cdot 10^{-7}$) of an app's download rank (e.g., "1K", "50K", "10M") and the time required to adapt a new library version.

Public API Compatibility Further, we were interested in whether the compatibility of the libraries' public APIs, through which app developers integrate libs, influences lib adaption. For quick adaption of a new library version, a stable public library interface for consecutive versions is helpful. Addition of new methods to the API is less problematic in this regard, while deletion of methods or changes in the signature (parameter or return types) might force app developers to adapt their code.

We analyzed all libraries in our database with at least ten consecutive versions for which

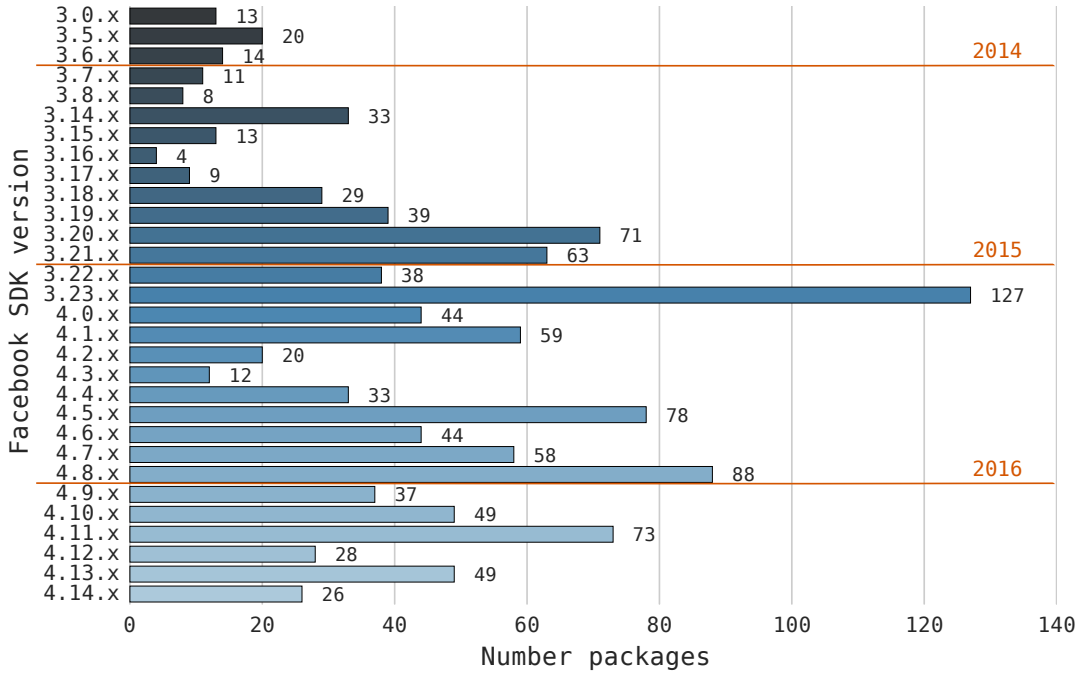


Figure 5.8: Distribution of Facebook SDK versions for the current top apps on Play. Orange bars indicate the beginning of the labeled year.

we also have the release dates. This comprises in total 70 distinct libraries. We define the public API to be stable between consecutive versions if there are at most additions of public methods but no deletions or modifications of existing signatures. The public API compatibility ranges from 7% (*Crittercism* lib) to 87% (*Amazon Analytics* lib). For this dataset, we could not detect any statistically significant correlation between the adaption rate of new lib versions and a lib's API compatibility (Kendall's $\tau = -0.29$ with $p = 0.2401$) or the changes in the lib's public API (Pearson's $r = 0.01$ with $p = 0.7637$), respectively. This warrants further investigation into the motivation for app developers to adopt lib updates and potentially other factors, such as library documentation (e.g., both *Facebook* and *Android's support v4* only have slightly above 50% compatibility on average between updates and they release detailed changelogs and even upgrade guides to support developers, but have quite different rates of adaption of their updates among the top apps).

5.7.2 Detecting Vulnerable Library Versions

We use LibScout to investigate the presence of vulnerable third-party libraries within our app repository. In particular, we use two highly severe vulnerabilities from the recent past as case studies: an account hijacking vulnerability in the Facebook SDK v3.15 [133] and CVE-2014-8889 of the Dropbox SDK versions 1.5.4–1.6.1 [49] that allowed attackers to capture the user's Dropbox files via vulnerable apps. For each

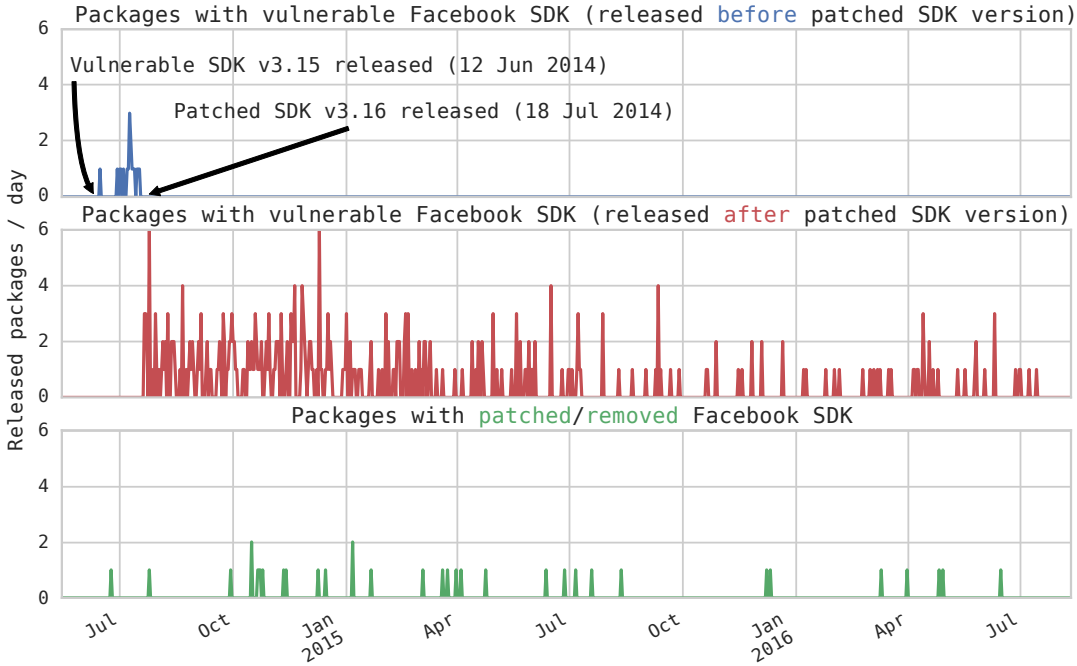


Figure 5.9: Daily releases of packages with a vulnerable or patched Facebook library between 04/2014 and 07/2016

incident, we investigated the number of affected packages/apps, their user-base, as well as vulnerability window and time-to-fix for the affected apps.

Facebook Account Hijacking Facebook released their SDK v3.15 for Android, which contained an account hijacking vulnerability, on 06/11/2014. In the histories of our sample set apps, we discovered, in total, 394 affected packages from 51 distinct apps, when only considering packages with exact matches of the vulnerable lib version. For 18 of those apps, we knew the number of downloads from Play and are able to estimate a lower bound of 69M downloads of vulnerable packages. For 356 affected packages, our dataset contained the release dates and enabled us to investigate the vulnerability windows and times-to-fix of those packages. Figure 5.9 illustrates the releases of vulnerable and fixed packages in our dataset, where we also consider removal of the Facebook lib from the app as a fix. Most noticeably, the majority of the vulnerable packages (338, in the middle facet of the figure) were released *after* Facebook released the fixed SDK v3.16, in some cases even still in July 2016, i.e., more than 1.5 years after the patched SDK. Of the affected apps, 13 apps never released a fixed version and even their latest version on Play is still vulnerable. For 33 of the remaining apps, we can calculate the average vulnerability window and time-to-fix with absolute certainty (i.e., no gap in release dates and version history) and their average time-to-fix is 188 ± 55 days and average vulnerability window is 190 ± 55 days. Those are worrisomely high numbers that expose the end-users to unnecessary long vulnerability periods when considering

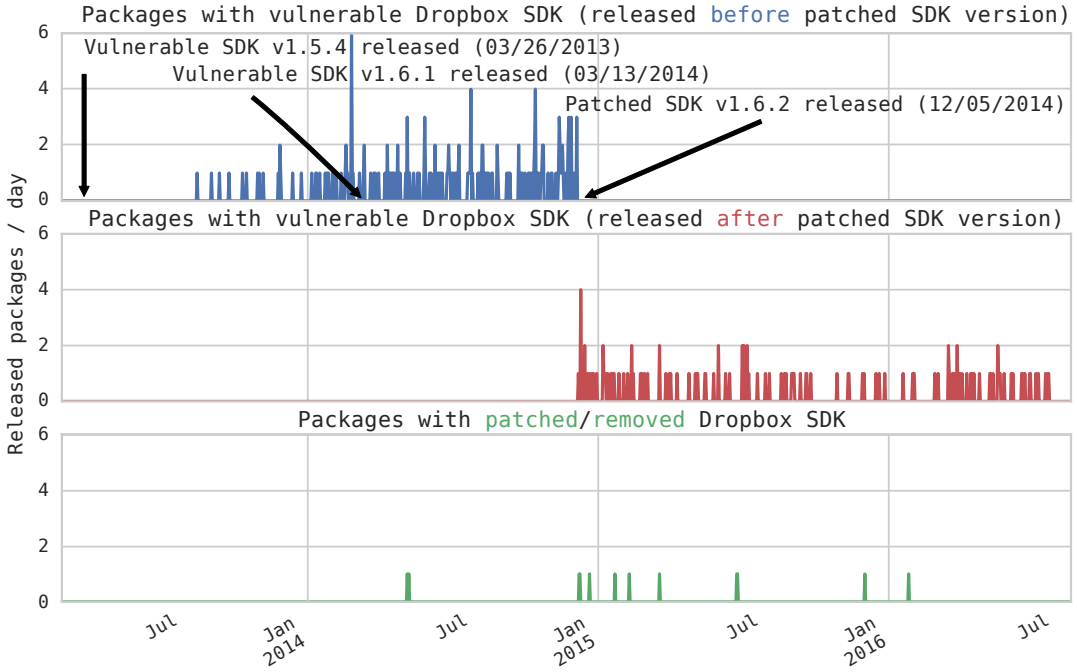


Figure 5.10: Daily releases of packages with a vulnerable or patched Dropbox library between 04/2014 and 07/2016

that Facebook released a fixed version only 36 days after the vulnerable version (see Figure 5.9).

Dropbox Data Stealing For the vulnerability of Dropbox SDK versions 1.5.4 and 1.6.1, we found in our sample set 360 affected packages from 23 distinct apps, when only considering exact library matches. For 11 affected apps our dataset contains information on their downloads, which accumulate to 11M. For 301 packages we have the release dates available, which allow us to investigate their time-to-fix and vulnerability windows. Figure 5.10 depicts the daily releases of patched/vulnerable apps with the Dropbox SDK and from our dataset we derive an average time-to-fix of 59 ± 110 days and an average vulnerability window of 196 ± 127 days due to three affected library versions prior to the patched SDK v1.6.2 (Dropbox fixed that vulnerability within one day after notification). Of the 23 affected apps in our dataset, 9 did not release a fixed version and their vulnerable versions are still the most recent ones on Play.

5.7.3 Analysis of Security-related APIs

Prior studies [52, 112] have shown that misuse of cryptography APIs is widespread among Android apps and that dynamic code loading by apps can be exploited to hijack apps. However, the original studies reported their results on a *per-app* basis and did not consider the extent to which third-party libraries contribute to this problem. Thus,

Property	#Libs/Ver	Verified
R1: ECB mode for encryption	5/25	5/25
R2: constant IV for CBC	7/32	4/20
R3: constant symmetric keys	13/60	3/7
R4: static salts for PBE	2/2	2/2
R5: <1000 iterations for PBE	2/2	2/2
R6: static seed for SecureRandom	3/7	2/5

Table 5.4: Results for crypto API analysis of ad libs showing candidate and verified libs/versions in our library set.

we were interested in how prevalent such misuse is among third-party libraries and how helpful techniques like LibScout could be to augment such studies with better accountability. We focus our analysis of security-related APIs on advertising libraries since they are the most popular and widespread type of libraries. For both analyses, we used WALA to create a set of candidate libraries by scanning the bytecode of the 315 ad samples of 39 distinct ad libs (see Table 5.2) for the relevant API calls. Since the number of samples is suitable for manual review, we refrained from re-implementing the original analysis methods and manually verified whether security properties were violated or not in our candidate libraries.

For the crypto API usage, we performed the same six checks as in prior work (R1-R6 in Table 5.4). This includes checks for constant encryption keys (R3), salts/seeds (R4+R6), and initialization vectors (IV, R2), as well as checks for the discouraged usage of ECB mode and a low number of iterations in password-based encryption (PBE). Table 5.4 summarizes our findings. The middle column shows the number of candidate libraries and versions based on the presence of the respective APIs. The last column shows the verified violations after manually checking the bytecode of each candidate. Out of all available ad libraries, 10 libs violate at least one of those properties. Several libs violate multiple properties, e.g., *Adrally* (R1,R2,R4,R5,R6), *Leadbolt* (R2,R3,R6), *domob* (R1+R2) and *AppFlood* (R4+R5). In 12 of the 18 verified libs (66%), all versions of those libraries were affected and in 14 cases even the latest available version in our dataset was still affected. All of the "fixed" libraries simply removed the affected code segment, but not a single one actually implemented a proper fix for the previously vulnerable code. The only library (*Leadbolt* SDK) that modified affected code, replaced an empty initialization vector (R2) in versions 5.x with a constant IV in version 6.0.

We used LibScout to detect the affected application packages in our dataset. In total 2,667 app versions of 296 distinct apps with a cumulative install-base of 3.7bn were affected by those ad libs with verified crypto misuse. In summary, improper usage of cryptography APIs is very common among the widespread ad libs and thus future work should investigate to which extent prior results [52] must be attributed to the library developers instead of app developers.

Second, we study dynamic code loading behavior of ad libs. We follow the approach described in [112] and test whether code loading is performed via package contexts, Runtime.exec, or the DexClassLoader. Only 9 out of 39 ad libs use any form of dynamic

code loading. In fact, only the *HeyZap* lib does code loading via the package context, but it only exposes this functionality to app developers. Only six libraries (33 versions) make use of Runtime.exec to execute logcat or some shell operations for modifying access rights of files. Only one version of the *ChartBoost* SDK contains a suspicious call to check for the presence of the superuser binary. The DexClassLoader technique is only used by the last two *Admob* versions and one version of *Tencent*. Both libs load a supplemental jar/dex file. In summary, dynamic code loading is not widespread in ad libs in our dataset and is rather attributed to other principals (e.g., app developer or other libraries).

5.8 Discussion

In Section 5.7 we studied security vulnerabilities in third-party libraries and, using LibScout, we were the first to show to which extent these problems actually affect the current set of top apps on Google Play. Parts of our analyses re-applied prior approaches [52, 112] and our results show that third-party libraries are a contributing factor to those original results, which reported only *per-app* results or could only exclude a small set of libs from their results based on unobfuscated library package names. Thus, we argue that a lightweight approach for third-party library detection, like LibScout, which is resilient against common obfuscation techniques, can greatly enhance static analysis approaches (e.g., [59, 151, 94, 19, 144, 63, 112, 106, 54]) by allowing them to attribute their results to the correct principals (app or library developers).

Moreover, our results show that app developers adopt new library versions only very slowly, even when the currently used version contains severe security or privacy vulnerabilities. While this chapter built the foundation to reliably detect library (versions) within binary application packages, Chapter 6 continues this work and seeks to answer questions about the root causes of the current status quo including different principals of the Android ecosystem, such as the app developer, library developer, or the marketplaces.

Current Limitations and Future Work The design choice to extract profiles from the original libraries comes with the inherent limitation of completeness. Therefore, complementary techniques such as *WuKong* [141] or *Libradar* [95] can be useful to detect potential libraries that are not in LibScout's database. Currently, our approach does not detect code-only changes in libraries, however, we can limit the number of candidate versions to a small set. To pinpoint the exact version whenever the profiles are ambiguous (see Section 5.6.1), we are experimenting with secondary, code-based profiles that are generated from dex bytecode operation types (e.g. invoke or move) after compiling the library jars to dex bytecode. These secondary profiles are still resilient to identifier renaming but could be influenced by code-based obfuscations such as API hiding. Identifying changes in the set of code instructions can also be leveraged to detect repackaged apps and piggy-backed apps, i.e. clones of popular applications that have been instrumented with malicious code. In [S3] we generate application profiles to

detect apps from the same application generator service that only differ in config files and resources. Prior work discovered that the malicious payload in piggy-packed apps is often triggered via static calls inserted into known libraries to automate hooking [86]. Code-reuse detection on instruction or basic block level can identify such additions—a strong indicator for piggy-backed apps.

Although being robust against a larger number of obfuscation techniques than related work, different kinds of code optimizations can negatively influence the similarity score. Techniques such as removal of unused method parameters modify the API signature, code inlining and vertical/horizontal class merging either remove methods completely or move classes into different packages. This affects the original code layout or removes existing APIs. Consequently, the similarity score drops, similar as for dead code elimination. More extreme obfuscations such as virtualization, class encryption (cf. Table 5.1), or package hierarchy flattening would still defeat our static approach (and any related approach). In the latter case, the structure of the hierarchy is modified and the boundaries between app and library code become blurred. Since such techniques introduce severe side-effects, they are rarely used in practice.

Another open problem is the manual or automatic dead-code elimination of library code. The complete library code is no longer statically included but only a (small) subset thereof. Our partial matching algorithm covers this problem up to the point where only so little library code is left in the app that the similarity score drops below a certainty threshold. In general, this is a hard problem that can not be completely solved with static analyses techniques. Instead, only solutions in which dependencies have to be declared explicitly would remedy this problem.

A few libraries, e.g., the Baidu SDK that recently had a severe vulnerability [132], are provided as a native library. Such libraries could be detected based on the hashes of their shared object files included in the app packages. Providing a database for native libraries would complement our approach.

Lastly, our repository contains only the top apps on Google Play. Thus, the results of our study might shift when we also consider the "long tail" of apps. We deliberately decided on this approach, since those top apps account for the bulk of all downloads and the largest user-base on Play. Another technical reason is that building complete app version histories is also a highly time-consuming task (in our experience, one Google account can query one app version per 2–5 seconds).

5.9 Related Work

There is a large set of literature targeting code cloning detection techniques. Davies et al. introduced the term *Software Bertillonage* [41, 60] to group different techniques to identify the provenance of a software component, such as graph matching, control-flow matching or signature-based identification. In their paper [41] they devise a technique called *anchored signature matching* to determine class provenance of source code within Java binaries. In contrast to our Merkle-tree based profiling, they do not account

for bytecode obfuscation and do not exploit package information to match the code structure. Ishio et al. [78] extend this signature matching by adding code-based features to refine their approach to detect reuse of software components. In contrast to LibScout, none of these approaches extracted their signatures from the ground truth in form of the original source/libraries. On Android, code and app clone detection techniques have been studied in many respects. Prior work identifies repackaged applications by computing similarity between apps using code-based similarity techniques [33, 70, 160] or by extracting semantic features from program dependency graphs [39, 40].

Other approaches are tailored to third-party libraries on Android, e.g., by employing the concept of whitelisting package names to detect libraries within app code [65, 29, 33]. As such approaches fail to cope with even simple obfuscation techniques such as identifier renaming, more robust approaches based on machine learning or code clustering have been investigated. *AdDetect* [102] and *PEDAL* [93] use machine-learning to detect advertising libraries. AdDetect uses hierarchical package clustering to detect (non-)primary modules of apps whereas PEDAL extracts code-features from library SDKs and uses package relationship information to train a classifier to detect libraries even when identifiers are obfuscated. *AnDarwin* [40] and *WuKong* [141] detect app clones with high accuracy by filtering library code that is detected by means of code clustering techniques. Such approaches rely on the assumptions that libraries are pervasively used by many apps, and app developers do not modify the library. However, this second assumption is unrealistic, since automatic/manual dead-code elimination during app building will necessarily modify the library code [6]. Moreover, these approaches only provide binary classifications since they cannot name the concrete library versions used within the apps. The recent *LibRadar* [95] extends WuKong's clustering approach and generates unique profiles for each detected cluster. Profiles are generated from the frequency of API calls within distinct packages in a cluster and can subsequently be used for fast library detection. With this approach, *LibRadar* was able to find 29K potential libraries on a large corpus of Google Play apps. This number presumably constitutes an over-approximation, since the original code clustering and the subsequent feature extraction are not performed on the original libraries. This lack of ground truth produces false positives when multiple libraries have the same root package (e.g. com.google for the various Google Play Service libraries and Google libs like Gson/Guice). To avoid such heuristics, we extract our profiles from the original library binaries. This comes at the cost of completeness but has several advantages such as less false positives and the possibility of inferring exact library versions. In addition, this allows computation of more reliable similarity values for partial library inclusions (as a result of code optimizations).

Besides library detection, research has also proposed techniques for privilege separation between host apps and advertising libraries. To this end, *AdSplit* [121] puts library code into separate processes, while *PEDAL* [93] uses an inline reference monitoring approach to allow users to selectively enable/disable functionality in libs that require a permission. *AdDroid* [111] uses a system-centric approach and proposes a new ad-API in Android's application framework for app developers to allow privilege separation by construction. As a more rigorous approach, *APKLancet* [150] removes malware and ad library code

from an app package. The code to be removed is identified through semantic fingerprints that have previously been extracted from malware/library samples.

A separate line of work studied questions related to security and privacy in third-party libraries. Stevens et al. [128] investigate the permission (mis-)use of advertising libs. Book et al. [29] conducted a longitudinal study of ad lib permissions and discovered that the absolute number of required permissions increases over time. However, in contrast to our longitudinal studies, they estimated the library release dates as the terminus ad quem of the release date of an app that includes the library. Library profiles are generated by hashing the library code detected via package name matching. In this case, the detection only works if the original library code is included without any modifications. Other approaches analyze private data exfiltration [65, 123] and security vulnerabilities in authentication/authorization SDKs [142]. Linares-Vasquez et al. [92] apply signature-based code matching to revisit prior studies in the context of code obfuscation and library usage. Their results are in line with our security analysis of crypto API misuse (see Section 5.7.3) in which we showed that libraries are a contributing factor to security issues. However, their work tries to distinguish library/obfuscated code from app developer code and does not seek to identify specific libraries (or versions) as an origin.

5.10 Conclusion

There is a trend to include more and more libs into apps, while at the same time app developers slowly adapt to new library versions (if at all). This puts millions of users at risk if security vulnerabilities remain unfixed in current top apps. Similar to the Android fragmentation problem, our results show strong indications for a library version fragmentation problem. Even in top apps, severely outdated library versions were found, implying that library providers can not act on the assumption that end-users may use the latest features or have the latest bugfixes. We continue this line of work in Chapter 6 to answer questions about the root causes and to measure to which extent libraries in current apps could be updated based on their library usage.

Ethical considerations. We reported our findings to Google's App Security Improvement (ASI) program [7].

6

Library Updatability

An Empirical Study of Third-Party Library Updatability

6.1 Motivation

Third-party libraries in Android apps have repeatedly been shown to be hazards to the users' privacy and an amplification of their host apps' attack surface. A particularly aggravating factor to this situation is that the libraries' version included in apps are very often outdated (cf. Section 5.7).

This work makes the first contribution towards solving the problem of library outdatedness on Android. First, we conduct a survey with 203 app developers from Google Play to retrieve first-hand information about their usage of libraries and requirements for more effective library updates. With a subsequent study of library providers' semantic versioning practices, we uncover that those providers are likely a contributing factor to the app developers' abstinence from library updates in order to avoid ostensible re-integration efforts and version incompatibilities. Further, we conduct a large-scale library updatability analysis of 1,264,118 apps to show that, based on the library API usage, 85.6% of the libraries could be upgraded by at least one version without modifying the app code, 48.2% even to the latest version. Particularly alarming are our findings that 97.8% out of 16,837 actively used library versions with a known security vulnerability could be easily fixed through a drop-in replacement of the vulnerable library with the fixed version. Based on these results, we conclude with a thorough discussion of solutions and actionable items for different actors in the app ecosystem to effectively remedy this situation.

6.2 Problem Description

Third-party libraries are an indispensable aspect of modern software development. They ease the developer's job, for instance, by providing commonly used functionality, sharing programming know-how among developers, enabling monetization of software, or integrating social media such as Facebook or Twitter. In contrast to the benefits that developers reap from third-party code, end-users of software are reportedly exposed to an increased risk to their privacy and security by those external software components. Recent reports [124, 72] warn of the hidden costs of libraries in form of buggy code that increases the app's attack surface and introduces security vulnerabilities. Sonatype [124] reports that older software components have a three times higher rate of vulnerabilities and that almost 2 bn software component downloads per year include at least one security vulnerability. These numbers are backed with findings from different software ecosystems, e.g., for Windows applications [101] and Javascript libraries [83]. Moreover, their results show that, although library updates with security fixes exist, they are not adopted by developers.

Similarly, recent works [P3, 34] have reported such alarming findings for the Android ecosystem. About 70% of all third-party libraries in apps are (severely) outdated and a slow adoption rate of updates of about one year aggravates the library outdatedness problem. As a consequence, fast response times by library developers remain noneffective

and even known security vulnerabilities [132, 134, 133, 49, 62] remain a persistent threat in the app ecosystem when app developers do not integrate the existing fixes into their apps. Google recognized their central role as market operator early for amending this problem and introduced their application security improvement program (ASI) [7] in 2015. In this ongoing effort, Google notifies developers when security problems were detected in their apps and/or included third-party components and enforces a remediation period to fix the detected vulnerabilities. According to their statistics [9], this approach already proved to be successful in improving the overall app market security. However, the main drawback is that this approach only fights the symptoms of the underlying problem of developers not keeping dependencies up-to-date.

6.3 Contribution

To improve on this situation more sustainably, for instance by realizing effective solutions that are practical and accepted by all involved parties, it is important to first understand the app developers' motivation for not updating third-party dependencies and to investigate the role of other actors—like the library developers—in the current situation. This work makes the first contribution towards such a solution by identifying the root causes *why* app developers do not update third-party libraries on Android. We start with conducting a survey with 203 app developers from Google Play to collect first-hand information about library usage in apps. Among others, this survey covers questions regarding library selection criteria, developer tools, reasons to (not) update, as well as feedback and comments on what app developers think needs to be changed to enable more effective library upgrades. These insights motivate a follow-up library release analysis that uncovers that library developers are very likely a contributing factor to the poor adaptation rate through an inconsistent and imprecise library version specification, i.e., the actual changes in code and API do not match the expected changes conveyed by the version numbers (*semantic versioning*). As a result, app developers cannot properly assess the expected effort for upgrading the library and ultimately abstain from an update to prevent ostensible effort and incompatibilities.

To investigate the actual effort of updating libraries, we conduct a large-scale library updatability analysis of 1,264,118 apps from Google Play. We analyzed the apps' bytecode to check whether included libraries are actually called by the app. Combining this data with the results of an analysis of each library's API robustness across its different versions, we determine that 85.6% of all libraries can be updated by at least one version, in 48.2% of all cases even to the most current lib version, simply by replacing the library and without the need to change the host app's code. Contributing factors for this high updatability rate are a generally low library API usage, i.e., on average 18 library API calls, and the fact that the most frequently used APIs remain stable for the majority of libraries. Focusing on security incidents, we find 16,837 actively used libraries in apps that contain one publicly known security vulnerability. Based on our analysis, 97.8% of these libraries could be patched by simply exchanging the vulnerable library with the fixed version, again without the need to change the app's code.

Finally, the results of the developer survey and our follow-up analyses helped us to identify problem areas and weak links in the ecosystem. In Section 6.7 we summarize our findings and propose actionable items for different entities including library developers, the marketplace, development tools, and the Android system to remedy the situation. Based on our findings and the responses from our survey, we believe that these solutions are both effective in amending the library outdatedness problem and accepted by the majority of developers. In summary, this work makes the following contributions:

(1) We conduct a survey with 203 app developers from Google Play to collect first-hand information on library usage and to identify root causes of developers not updating their dependencies.

(2) We analyze library releases to uncover that library developers are likely a contributing factor to a poor library adaptation. In 58% of all library updates, the expected changes derived from semantic versioning do not match the actual library code changes.

(3) We conduct a large-scale analysis of 1,264,118 apps to identify libraries and their API usage. In 85.6% of cases, the detected library can be updated by at least one version, in 48.2% of cases even to the most current version. In addition, we find 16,837 apps that include a library with a known security vulnerability, out of which 97.8% could be patched without app code adaption.

(4) Finally, we thoroughly discuss short-/long-term actionable items for different entities of the app ecosystem to remedy the problem of outdated libraries.

6.4 App Developer Survey

We conducted an online survey with Android application developers who already published at least one application on Google Play. We investigated the developers' main motives and knowledge when it comes to managing third-party libraries for their apps. Mainly, we were interested in the following three questions:

Q1: What is the common workflow to search for and to integrate third-party libraries into applications?

Q2: How frequently do developers update their apps/libs and what is their main motivation for updates?

Q3: What are possible reasons to not update dependencies and what solutions could app developers think of?

6.4.1 Ethical Concerns

The questionnaire (see Appendix A) was approved by the ethical review board of our university. We also took the strict German data and privacy protection laws into

Table 6.1: Demographics of developer survey participants.

Gender		**Age** ($\overline{x}$ = 32.90 ± 1.60 **years**)	
Female	10 (04.93%)	15–19	6 (02.96%)
Male	186 (91.63%)	19–29	63 (31.03%)
No answer	7 (03.45%)	29–39	64 (31.53%)
		39–49	31 (15.27%)
Highest educational degree		49–59	15 (07.39%)
Graduate	117 (57.64%)	59–69	4 (01.97%)
College	41 (20.20%)	No answer	20 (09.85%)
High school	30 (14.78%)		
No degree	12 (05.91%)		
No answer	3 (01.48%)		

account for collecting, processing, and storing participant information. We collected email addresses from Android application developers who had previously published at least one application on Google Play and kindly asked them to participate in our online questionnaire, whether they like to be blacklisted for future user studies, and whether they want to learn more about our scientific work. Overall, we sent out 60,000 invite emails. Before filling out the questionnaire, developers had to consent to the use and publication of their answers.

6.4.2 Participants

In response to the invitation emails, 203 app developers finished the questionnaire within five days (participation rate of 0.34%). Of all participants, 91.6% reported being male, 4.9% female, and the remaining 3.4% declined to answer. Participants' mean age was 32.9 years (with a margin of error of 1.6 years with α = .05). The general coding experience was relatively high with a mean of 12.11 ± 1.35 years. The Android experience was reported with 4.06 ± 0.33 years on average. Of all participants, 34% affirmed that developing apps is their primary job. Asked about the context of app development, 35.5% reported to develop apps in a company, 38.4% are self-employed, and 61.6% develop apps (also) as a hobby. The participants reported having worked on 13.188 ± 4.42 apps. A detailed overview of the participants' demographics and professional background can be found in Table 6.1 and Table 6.2.

6.4.3 Q1: Workflow and Integration

In the first part of the survey we seek to answer how app developers choose and integrate libraries into their apps. Figure 6.1 shows the primary sources of the participants to search for libraries. It is evident that the majority of app developers use search engines, followed by the project hoster GitHub. The relatively small number of dedicated Android community websites, such as *Android Arsenal* or *Android Weekly*, underlines the lack of a central library marketplace/package manager such as *Cocoapods* for iOS or *npm* for JavaScript. Being asked about library selection criteria (see Figure 6.2),

Table 6.2: Professional background of participants in our online app developer survey.

Years of general coding experience		**Years of Android experience**	
< 1 year	4 (01.97%)	< 1 year	2 (01.0%)
1–5 years	45 (22.17%)	1–2 years	19 (09.5%)
5–10 years	51 (25.12%)	2–3 years	28 (14.0%)
10+ years	103 (50.74%)	4–5 years	40 (20.0%)
$\overline{x}$ = 12.11 ± 1.35 years		5–10 years	37 (18.5%)
		10+ years	4 (02.0%)
		$\overline{x}$ = 4.06 ± 0.33 years	
How learned Android programming[†]		**Developing apps primary job**	
Self-taught	182 (89.66%)	Yes	69 (33.99%)
On the job	66 (32.51%)	No	134 (66.01%)
Online coding course	38 (18.72%)		
Class in university	25 (12.32%)	**Context of app development**[†]	
Class in school	8 (03.94%)	Company	72 (35.47%)
Other	0 (00.00%)	Self-employed	78 (38.42%)
		Hobby	125 (61.58%)
Number of apps worked on		**Company size**	
1–5 apps	101 (50.00%)	< 10 employees	27 (37.50%)
6–10 apps	50 (24.75%)	10–50 employees	16 (22.22%)
11–50 apps	44 (21.78%)	50–100 employees	7 (09.72%)
51–100 apps	5 (02.48%)	100+ employees	22 (30.56%)
100+ apps	2 (00.99%)		
$\overline{x}$ = 13.188 ± 4.42 apps			

[†] Multiple choice, sum does not need to equal 100%

79.7% of all participants named functionality as main criteria. Open source (61.7%) and good documentation (52.3%) are further criteria for library selection. In general, recommendations and user ratings are less important. Security (26.6%) and particularly the use of permissions (29.7%) are among the least important criteria, which is particularly surprising after news reports and scientific research on permission misuse of advertisement libraries [134, 65, 128, 29].

Besides information about how libraries are chosen, it is important to know the preferred development platform and integration approach by developers. Figure 6.3 suggests that *Android Studio* is the preferred IDE for app development (61%), followed by (multi-platform) application generator frameworks such as *Xamarin* or *Cordova* (17.2%) and *Eclipse* with the Android plugin (13.3%). A small fraction of app developers (8.4%) prefers different environments such as *NetBeans* or even the command line. Similar to development platforms, there are different possibilities to integrate a library (see Figure 6.4). The Android *Gradle* plugin, introduced in 2014, is a powerful dependency manager and the default in Android Studio. Although two-thirds of app developers use Gradle, more than half of them also resort to manual inclusion or use a combination of different approaches. Build systems such as *Maven* (14%) or *Ant* (3.9%) are not widespread in Android app development. Users of *Xamarin* prefer to use its convenient package manager *NuGet*.

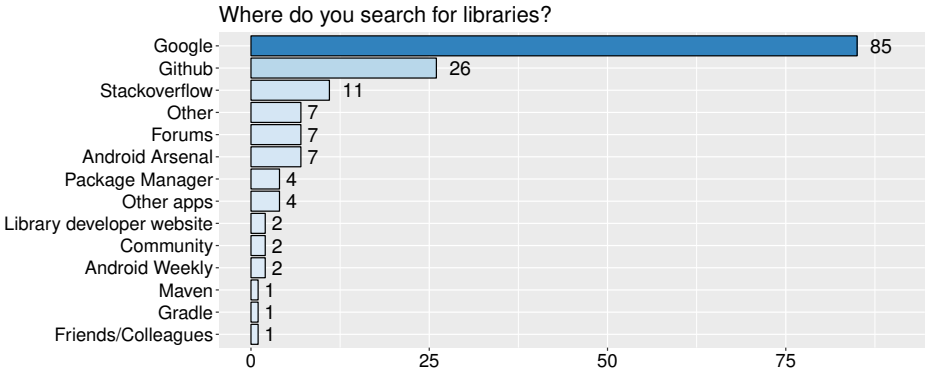


Figure 6.1: Primary sources for finding libraries among our survey participants

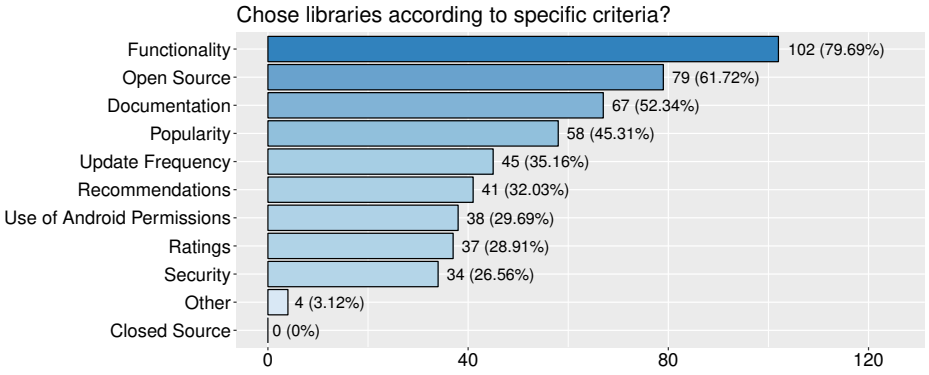


Figure 6.2: Reported criteria for library selection among our survey participants

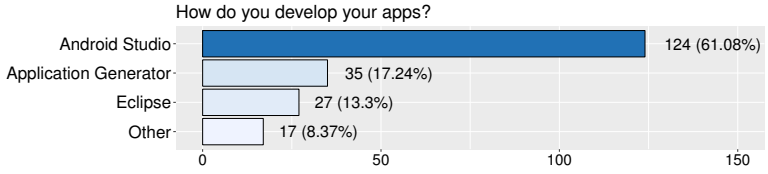


Figure 6.3: Primary development environment of our survey participants

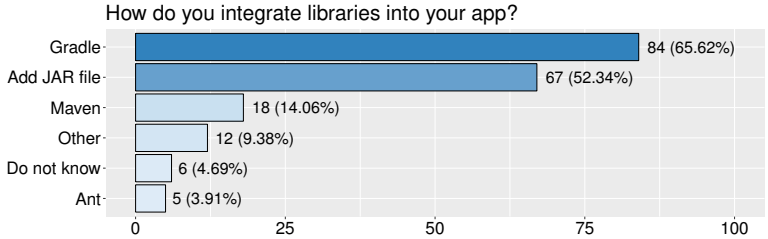


Figure 6.4: Used library integration techniques by our survey participants

6.4.4 Q2: Application and Library Maintenance

In the second part of the survey, we asked the participants about app release frequency, whether they update their dependencies, and about their main motivation to perform app and library updates (cf. Figure 6.5). 78% of the app developers release new app updates on a variable schedule, while only 22% rely on a fixed schedule, e.g., developers at companies with a fixed release schedule. The majority of developers releases new updates within a time period of one to three months. However, there is also a considerable number of developers (39.9%) that provides updates at most twice a year.

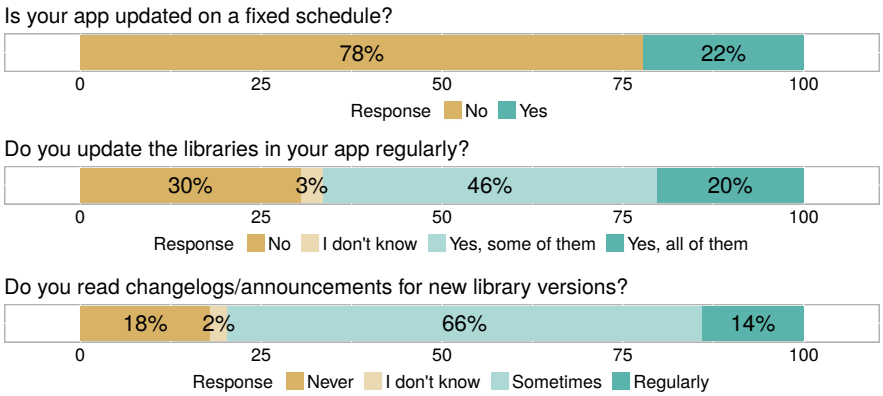


Figure 6.5: Questions and responses for Q2 regarding app/library release frequency.

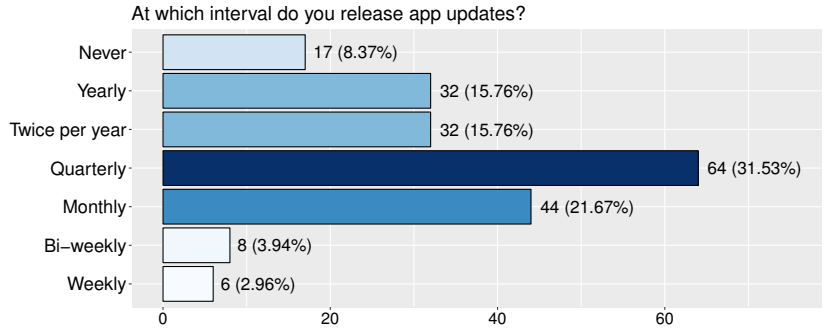


Figure 6.6: Interval at which our participants release their app updates

The main motivation to release new app versions is to provide new functionality and fixing bugs (see Figure 6.7). Only one-third of the developers explicitly names library updates as a reason to provide a new app version. This is contrary to the main motivation to update the apps' libraries where the dominant answer is bug fixing (only three developers did not name this). Functionality is only the third most common reason (56.5%), right behind security fixes (57.6%). Of all app developers, 66% update at least some of their libs regularly, while 30% completely abstain from updating the dependencies. Changelogs and release announcements are an effective means to reach app developers since 70% of the developers read them at least sporadically.

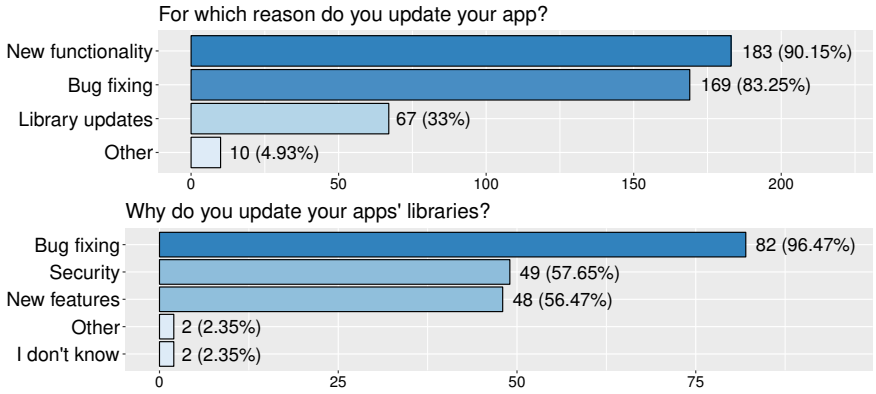


Figure 6.7: Reasons why our survey participants update apps and their apps' libraries

6.4.5 Q3: Reasons for Outdated Libs

The last part of the survey asked questions about problems that might be a reason for not updating libraries. We also requested a self-reporting on reasons for outdated libraries and asked the developers for their opinion on possible solutions.

Since Gradle is the default dependency management system in Android, we asked about Gradle's usability and drawbacks. While the majority of the participants likes Gradle (64.3% in Figure 6.8) or only sees minor limitations (31%), only three participants are unhappy with Gradle's usability. The most frequently named drawbacks include a weak build performance with more complex apps and a steep learning curve compared to the simplicity of adding libraries manually. We then explicitly asked for reasons that their apps contain outdated libraries (see Figure 6.9). For 57% of the participants, there is no incentive to update the library as it works as intended. Half of the participants are afraid of experiencing incompatibilities, for instance, through modified or renamed library APIs, or they refrain from updating due to an expected high integration effort. Another reason is that app developers are just unaware of library updates (33%).

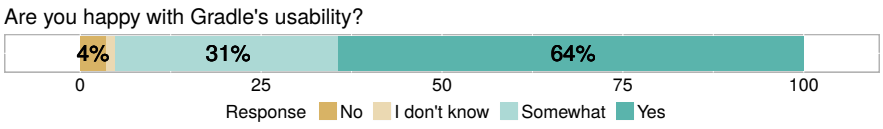


Figure 6.8: Usability satisfaction of our participants with the Gradle build system

Figure 6.10 shows a selection of potential approaches to facilitate better library management. Of all participants, 65.6% wish to have better development tools, for instance, an improved IDE integration. Among the app developers, 78.9% like the idea of having a central library marketplace or package manager, similar as in other ecosystems, such as iOS or JavaScript. Many library developers distribute their libraries via different channels, such as *Maven Central* or *Bintray*. For those who host their library only on their website, developers would welcome additional, potentially more convenient, distribution channels.

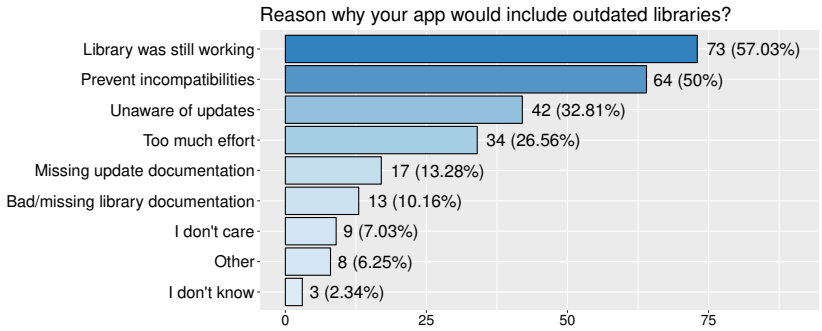


Figure 6.9: Self-reported reasons why the participants' apps would include an outdated library

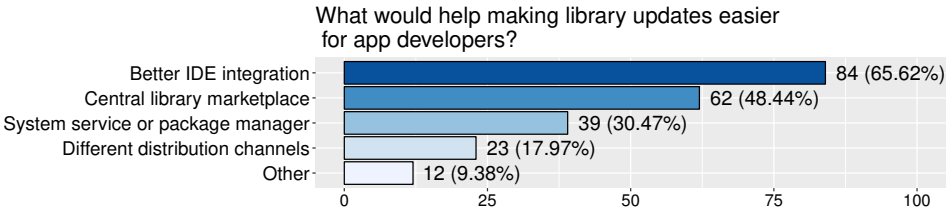


Figure 6.10: Preferred improvements for making library updates easier

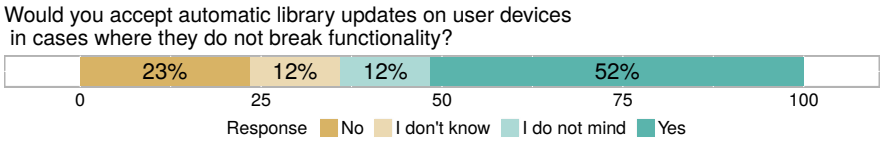


Figure 6.11: Acceptance of automatic library updates on end-user devices among our participants

Finally, we asked whether participants would accept an automated on-device library patching via the Android OS, as long as it would not break app functionality (cf. Figure 6.11). Half of the responses fully agreed with such a solution, while about 12% were not sure whether this is a good idea. About 23% clearly disagreed with such an approach, while another 12% did not mind.

6.4.6 Limitations

As with any user study, our results should be interpreted in context. We chose an online study because it is difficult to recruit Google Play developers for an in-person study at a reasonable cost. Choosing to conduct an online study gave us less control over the recruitment process; however, it allowed us to recruit a large and geographically diverse sample. Because we targeted Google Play developers, we could not easily take advantage of services like Amazon's Mechanical Turk or survey sampling firms. Managing online study payments outside such infrastructures is very challenging; as

a result, we did not offer compensation and instead asked participants to generously donate their time. As might be expected, the combination of unsolicited recruitment emails and no compensation led to a strong self-selection effect, and we expect that our results represent Android developers who are interested and motivated enough to participate.

In any online study, some participants may not provide full effort, or may answer haphazardly. In this case, the lack of compensation reduces the motivation to answer in a constructive manner; those who are not motivated will typically not participate in the first place. We attempt to remove any obviously low-quality data (e.g., responses that are entirely invective) before analysis, but we cannot discriminate perfectly.

6.5 Library Release Analysis

The survey results indicate that 77% of app developers update at most a strict subset of their included libraries (see Figure 6.5). One of the main reasons for this is that there is no obvious need to update the library when it works as intended. The survey suggests that bugfixes and security fixes would be a reason to update if new library versions would provide dedicated patch-only changes and would not mix bugfixes with new functionality. Another more alarming reason is that libraries are not updated due to the fear of experiencing incompatibilities and an expected high integration effort. This raises the question *how* library developers release new versions and whether their current release strategy could be a contributing factor to poor library adoption. In the following, we seek to answer this question by analyzing how often library versions change existing APIs and provide versions with mixed types of changes, i.e., security fixes and new functionality. A related but previously uncovered aspect is how library developers communicate these changes, i.e., which changes might an app developer expect given a library version number and do these expectations match the actual changes made in code and API.

6.5.1 Semantic Versioning

The concept of classifying a version number into different categories to infer the expected effort of integration was proposed as *Semantic Versioning* (SemVer) by Preston-Werner [115]. It comprises a set of simple rules that dictate how *library* developers assign and increment new version numbers. The basic idea is that if library developers adhere to these rules, the library consumer (typically the app developer) can assess, just by looking at the version string, whether or not a library update can be performed without additional implementation and code adaption effort. *Semantic Versioning* works as follows: First, the lib developer declares the public API, e.g., by documenting it. Then, any changes in the documented public API are communicated with the version number. The version format consists of three numbers X.Y.Z (Major.Minor.Patch). Whenever a new version includes bug fixes or code-only changes that do not affect the API, the patch version number is incremented. Backwards compatible API additions/changes

increment the minor version and backwards incompatible API changes (removed methods, incompatible argument types) increase the major number. Intuitively, a library without further dependencies can be updated without additional effort if a new version is a minor/patch version. A major version might require additional integration effort, depending on the changes made to the APIs in use.

6.5.2 Android Library Versioning

To investigate the status quo in Android library versioning we conduct an empirical study of expected changes versus actual changes to confirm or disprove that library developers can be a contributing factor to the problem of a poor library adaptation in the Android app ecosystem. To this end, we build on and extend our library database (cf. Section 5.5.1). In total, we analyze 89 distinct libraries with 1,971 versions with a minimum set of 10 versions per library. In our test set all libraries make use of the X.Y.Z versioning scheme, except *OrmLite* which uses an X.Y scheme. In addition, *Dropbox* (v2.0.5.1) and *FasterXML-Jackson* (v2.4.1.1) include a single library version with a sub-patch level. However, due to the absence of a changelog for these versions, we can not properly assess the necessity of such version numbers. In the following, we describe in more detail how we determine the actual changes in code and the expected changes conveyed by the version number.

Expected Changes We extend LibScout and integrate a version parser that classifies changes expressed by the version string into patch, minor, and major releases. By comparing consecutive library versions we then retrieve a list of expected changes, e.g., a version 2.4.1 immediately following version 2.3.7 is classified as a minor release.

Actual Changes Semantic Versioning requires that the public library API has to be properly defined at some point, either via an explicit documentation or via the code itself. Since some libraries either lack a full documentation or do not provide a history of their API reference, we programmatically extract the public API from the original library SDKs. The public API set of the first version of each library in our dataset is used as a baseline.

1. Filtering undocumented APIs: Undocumented public methods are not meant to be used by an app developer and hence should not be considered part of the public API. By extracting the public API programmatically, we have to filter such methods in a best effort approach (see also discussion in Section 6.7). To this end, we exclude public methods that reside in subpackages named internal. Moreover, we conservatively filter classes (and their declared methods) that have been renamed and shortened through an obfuscation tool like ProGuard [68]. Concretely, we consider classes named with one or two lowercase, alpha characters as obfuscated (following ProGuard's renaming rules).

2. Determining actual changes: To determine actual changes between consecutive library versions, we implement an API diff algorithm that operates on two sets of public

APIs api_{old} and api_{new}, where api_{old} is the API set of the immediate predecessor version of api_{new}. An API is described by its signature that includes package and class name as well as the list of argument and return type, e.g. com.facebook.Session.getAccessToken ()java.lang.String . If $api_{old} = api_{new}$ we have a *patch*-level release, i.e., there are code changes only. If $api_{old} \subsetneq api_{new}$, new APIs were added but existing ones did not change. This is classified as a backwards-compatible *minor* release. Whenever api_{old} includes APIs that are not included in api_{new} we conduct a type analysis to check for compatible counterparts in api_{new}. Compatible changes include generalization of argument types, e.g., an argument with type ArrayList is replaced by its super type List . Generalization on return types is generally not compatible and depends on the actual app code that uses the return value. Since we do not conduct a code analysis we treat non-matching return types as incompatibility. To not suffer from false positives, we furthermore abstain from searching for alternative candidates whenever the class and/or package name do no longer match, since this may result in ambiguity.[1] Hence we report conservative numbers when searching for API alternatives. If we are able to identify alternatives for all APIs that do not match exactly, we may classify the release as patch or minor. All other cases are classified as a major release.

6.5.3 Semantic Versioning Statistics

Applied to our library set, we found that in 58% of all version changes, the library developer incorrectly specified the new version string, i.e., according to Semantic Versioning rules the expected release level did not match the actual release level. Even worse, there is no single library that achieved a 100% correctness in versioning. Only 3/89 libraries (3.4%) correctly classified the release level in more than 80% of all releases, with the *android-oauth-client* library ranking first (93.8%). On the other hand, 10/89 (11.2%) of libraries specified the version string correctly in less than 20% of all cases. Two libraries (*universal-image-loader* and *log4j*) have not specified a single version change correctly. Further, we could not find a positive or negative, statistically significant, correlation between library category (e.g., Advertising, Utilities) and the Semantic Versioning classification score.

A mismatch between expected and actual changes is always disadvantageously for the library consumer in that she can not properly assess whether a new library version can be used as a drop-in-replacement or whether a considerable amount of work has to be spent to integrate the update. The severity of the mismatch, however, depends on the type of inconsistency. In particular, two types of inconsistencies are problematic: if either patch or minor release is expected, but the actual changes indicate a major release (highlighted cells in Table 6.3). These numbers show that library developers under-specify changes in 39% of all cases, i.e., the version increment is too moderate and suggests compatibility although API changes might break existing applications. In about 6.5% of cases, library developers over-specify changes. This does not affect compatibility but might impede wide-spread adaptation due to an expected high integration effort.

[1] Updating imports is typically done automatically by an IDE like Android Studio and is therefore not considered as incompatibility.

Table 6.3: SemVer misclassification by type (expected vs. actual change). Highlighted cells are critical as the actual semantic versioning suggests compatibility although the opposite is the case, i.e., the developer underspecified changes.

		Expected		
		patch	**minor**	**major**
Actual	**patch**	—	5.7%	0.85%
	minor	11.93%	—	0.48%
	major	**15.02%**	**24.02%**	—

Figure 6.12 summarizes the total number of expected and actual changes between consecutive versions by release level for the 1,971 analyzed versions. The expected changes denote how library developers specified new version strings. This distribution depicts what is expected for a typical library lifecycle; a stable base API with occasional additions and code-only changes such as bug and security fixes for the majority of releases. However, the reality looks different: 44% of all versions in our analysis were classified as major release due to non-compatible changes and/or removals of existing APIs. This indicates a poor library design without carefully taking into account the effort/incompatibilities that consumers might experience.

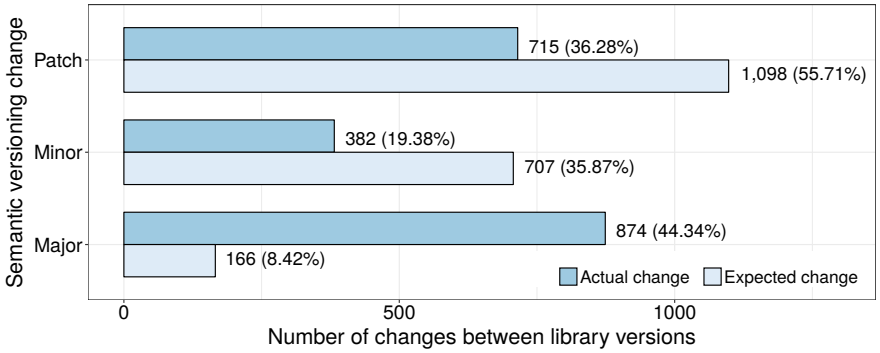


Figure 6.12: Total number of expected and actual changes between consecutive library versions grouped by patch/minor/major.

6.5.4 Security Fixes

Finally, we have a dedicated look at security fixes in libraries. These are the most important kind of updates and should typically be provided as a patch release. However, even when released with a short bug-fixing time, such patches miss their intended effect if they are slowly adapted by app developers or not at all. Ultimately, the end-user will be at risk and suffer from vulnerabilities like identity theft or private data leakage. To check if library developers adhere to this rule, we analyze the *Facebook* and *Dropbox* vulnerabilities used in Section 5.7.2, vulnerabilities in *Apache Commons Collections* (Apache CC) and *OkHttp* found via blog entries, as well as known library vulnerabilities reported by Google's ASI program [7]. In total, we were able to investigate eight distinct

Table 6.4: Library versions with a fixed security vulnerability, the expected and actual SemVer of the patch, whether and how the security fix is described and whether this library vulnerability is listed in Google's ASI program. Versions marked with (B) denote backport patches.

Library	Fix	exp./act. SemVer	Changelog	CVE	other	in ASI
Airpush	8.1x	minor / patch	–	–	–	✓
Apache CC	4.1	minor / **major**	security	–	blog+report	–
	3.2.2 (B)	patch / patch	security	–	blog+report	–
Dropbox	1.6.2	patch / patch	bugfix	2014-8889	blog	–
Facebook	3.16	minor / **major**	bugfix	–	–	–
OkHttp	3.2.0	minor / **major**	bugfix	2016-2402	blog	–
	2.7.5 (B)	patch / patch	bugfix	2016-2402	blog	–
MoPub	4.4.0	minor / **major**	bugfix	–	GitHub	✓
Supersonic	6.3.5	patch / **major**	–	–	–	✓
Vungle	3.3.0	minor / **major**	–	–	blog	✓

bugfix versions[2]. For the eight vulnerable libraries, we first determine whether the bugfix version is a patch release or whether the library provider mixed bugfixes with new content or even changed existing APIs. We subsequently compare these findings with the official changelog to see whether the fix is mentioned and properly documented. Table 6.4 shows the detailed results.

Six out of ten library patches (including two backports) are minor releases, i.e., the developer did not intend to provide a dedicated bugfix version. Only *Airpush* and *Dropbox* provide a patch-level fix, while *Facebook*, *MoPub*, *Supersonic* and *Vungle* provide a major version, i.e., they include new functionality and/or break existing APIs. *Apache CC* and *OkHttp* provide an additional backport of the security patch to allow an effortless adaption by older versions. Surprisingly, both backport versions are patch-only updates, while the fixes for the current releases were announced as minor versions and even included major changes. Mixing critical security patches with API changes is considered bad practice and does certainly contribute to a poor adaptation rate. Besides the version number, the changelog is the primary way to convey and explain important fixes and changes to the library consumer (see Figure 6.5). However, only *Apache CC* explicitly mentions a security fix in its changelog, four libraries at least mention a bug fix. Only the *Dropbox* and *OkHttp* vulnerabilities have a CVE entry. In order to provide transparency and increase the chance that the patch is adapted by developers, some libraries provide a blog/support entry in which they provide additional details about the vulnerability. *MoPub* at least provides a short note in its GitHub repository, referencing the respective ASI support document.

Although we cannot provide the same detailed analysis for the native libraries listed in the ASI program (*libjpeg-turbo*, *libpng*, *libupnp*, *OpenSSL*, and *Vitamio*), we checked their expected SemVer and changelogs for the fix versions. Only *Vitamio* provided the security fix as part of a major release (5.0). All other libraries provide a patch-level

[2]We had to exclude further vulnerabilities reported by ASI since we were not able to retrieve the original SDKs for either the fixed version and/or some older versions.

version and, more interestingly, even provide detailed changelogs for every (security) bugfix made. In our database of Java/Android libraries, only the Android support libraries and *OkHttp* provide comparable changelogs regarding the level of detail.

6.6 Library Updatability

Keeping third-party dependencies up-to-date is a complex problem with many facets and different actors involved. On the one hand, there are app developers who mainly wish to update libraries for bugfixes and security fixes (see Section 6.4). On the other hand, there are library developers that want app developers to adopt new library versions within a reasonable time-frame, e.g., for fixes and/or new functionality. In Section 6.5 we showed that library developers contribute to the adaptation problem by not giving app developers a simple means of assessing whether or not a new library version can be integrated without compatibility issues.

To properly assess the current status quo in library updatability, we analyze which library versions, and which parts thereof, are in use by applications. Given this information, we can then determine whether an actual major library release indeed requires additional integration effort or could still be updated as the set of used APIs remains compatible. To this end, we scan 1,264,118 apps from Google Play and identify included library versions. For each found library, we subsequently analyze the application bytecode to determine how the library is used in terms of API calls. Based on that information we infer the highest library version that is fully API compatible for that app/library combination.[3]

6.6.1 Approach

We extend the implementation of LibScout and implement the following analyses to conduct the updatability measurement study:

1. Library API robustness: We first analyze the robustness of the public library API across versions of a given library. For each library with more than 10 versions, we determine, on a per-API level, the highest version that provides this exact API. We are conservative in that we do not search for alternative candidates if the API in question is no longer available, e.g., due to method removal or renaming. Similar to the SemVer analysis, we filter methods that are obfuscated or reside in internal packages. As a result, we receive a comprehensive data set with updatability information for each library version/API pair. This analysis is much more fine-grained than the API compatibility analysis conducted in Section 5.7.1, which checks whether or not the *entire* API set of some version is present in the successor version. This data would be insufficient to determine whether an actively used library can be updated.

[3] There might be cases in which a fully API-compatible library version still breaks the client application. This is discussed in Section 6.7.3

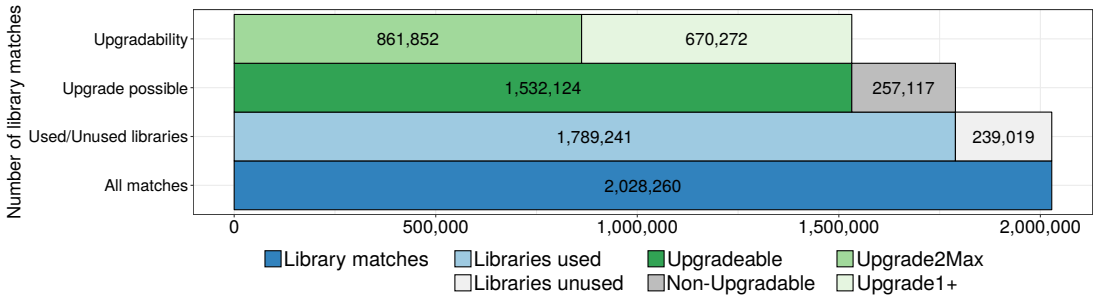


Figure 6.13: Library updatability of current apps on Google Play

2. Library usage: To identify the actively used parts of a library, we scan the application bytecode for invocations of this library. To account for identifier renaming obfuscation, we match the library API with the identified root package name, e.g., when the original library root package com.gson was obfuscated/renamed to com.mygson or com.ab, we rename the original library API accordingly. For ambiguous profile matches, i.e., LibScout is not able to distinguish patch-level changes in libraries, we select one of the matched libraries. Since patch-level changes are API-compatible this does not affect the subsequent updatability check.

3. Library updatability: Finally, we combine these two data sets to determine whether and to which extent libraries in apps can be updated. While libraries can, by definition, be replaced by patch and minor releases, this large-scale analysis investigates whether libraries can be replaced by subsequent major versions that account for 44% of all library releases. Furthermore, this allows us to identify hotspot-APIs, i.e., APIs of the libraries that are most/least frequently used, and to determine their stability.

6.6.2 Updatability Statistics

We conduct a large-scale evaluation in which we analyzed 98 distinct libraries and scanned 1,264,118 apps from Google Play. The results are summarized in Figure 6.13. LibScout successfully identifies 2,028,260 libraries (exact matches only). In 239,019 cases (11.8%), we could not detect any library APIs that are actively used, i.e., those libraries are dead code. For the remaining 1,789,241 libraries, we can determine the set of used APIs and correlate it with the API robustness data. The results suggest that in 85.6% of the cases the identified library can be upgraded by at least one version (*Upgrade1+*) without any code adaption, simply by replacing the old library. Even more surprising, a subset of 861,852 libraries (48.2%) can be upgraded to the most current library version (*Upgrade2Max*). Only in 14.4% of the cases, the library can not be upgraded by a single version without additional effort (*Non-Upgradable*), i.e., the next version changed or removed used APIs. One major reason for this high updatability rate is that although the majority of libraries offer hundreds or even thousands of different API functions, the typical app developer only uses a small subset thereof. Our results indicate that the average number of APIs used across libs is 18.

Table 6.5: Updatability to the most current version by sum of libraries and library matches grouped into 20% bins. How to read: Between 80–99% of all identified versions of 10 distinct libraries can be upgraded to the latest version. These 10 libraries account for 579,294 library matches.

Percentage	by # of libs	by # of lib matches
100%	5 (13.5%)	11,346 (1%)
80–99%	10 (27%)	579,294 (51%)
60–79%	5 (13.5%)	139,189 (12.3%)
40–69%	5 (13.5%)	121,671 (10.7%)
20–39%	4 (10.8%)	228,393 (20.1%)
0–19%	8 (21.6%)	55,690 (4.9%)
Total	37	1,135,583

In the following, we analyzed the extent to which libraries in apps could be upgraded to the *latest* version. In contrast to the SemVer analysis in the previous section, this puts a higher focus on the robustness of more popular APIs. Libraries that are stable in their most frequently used APIs, even across major versions, are assumed to have a high-updatability rate. To verify this assumption, we grouped 37 libraries for which we have more than 10 versions and more than 50 matches in our large-scale analysis according to updatability to the newest version. Table 6.5 shows the fraction of library matches that can be updated to the latest version, bucketed into 20% bins. The column on the right aggregates the absolute numbers of matches for those libraries.

We could not find a correlation between the absolute number of used APIs and library updatability. While the libraries in the top bucket on average use 11.9 APIs (with a standard deviation of $\sigma = 7.8$), the libraries in the last bucket only have a slightly higher API usage (mean = 14.7, $\sigma = 8.4$). However, aggregating the top ten most frequently used library functions and correlating them with their stability across library versions revealed the root cause. Libraries with a high updatability to the most current version are stable for the most popular APIs (even across major versions), while libraries with a very low updatability showed a completely different picture. In these cases, either (parts of) the most popular APIs have been completely replaced by a new APIs or existing ones have been modified or renamed. In our data set, Google's *Gson* library was a positive example with a 99.91% updatability to the most current version in 315,079/315,371 library matches. On the other hand, *Retrofit* could have been upgraded to the latest version only in 0.07% (20/30,568) cases due to major API changes in recent versions.

6.6.3 Security Vulnerability Fixing

Besides general library updatability, we are particularly interested in how easy vulnerable library versions can be patched. To this end, we investigate the eight publicly known vulnerabilities described in Section 6.5. We could not easily increase the set of libraries since it is not trivial to find reports on SDK vulnerabilities (see Table 6.4). However, the affected libraries are commonly used by many applications (14.4% of 1,264,118) and

Table 6.6: Number of apps found with a vulnerable library version, number of apps that actively use this library, number of apps that could be patched to the first non-vulnerable version without code adaptation (update2Fix), to the most current version available (update2Max), or not updated to a fixed version without code modification (non-fixable). Unused libraries are not considered in the last three columns.

Library	Versions	Found	inUse	update2Fix		update2Max		non-fixable	
Airpush	8.0	4,746	4,545	4,545	(100%)	4,545	(100%)	0	
Apache CC	3.2.1/4.0.0	1,199	749	749	(100%)	502	(67%)	0	
Dropbox	1.5.4–1.6.1	710	682	410	(60.1%)	6	(0.01%)	272	(39.9%)
Facebook	3.15	1,839	1,808	1,792	(99.1%)	4	(0.22%)	16	(0.88%)
OkHttp	2.1.0–2.7.4	7,319	7,179	7,169	(99.9%)	3,013	(42%)	10	(0.14%)
	3.0.0–3.1.2	500	237	237	(100%)	236	(99.6%)	0	
MoPub	3.10–4.3	—	—	—		—		—	
Supersonic	5.14–6.3.4	1,198	905	905	(100%)	743	(82.1%)	0	
Vungle	3.0.6–3.2.2	886	732	653	(89.2%)	594	(81.1%)	79	(10.8%)
Total		18,397	16,837	16,460	(97.8%)	9,643	(57.3%)	377	(2.2%)

thus any vulnerability in these libraries does affect thousands or even millions of users.

Table 6.6 shows the libraries with the range of vulnerable versions. After scanning the app repository, we found 18,397 apps that include one of the vulnerable library versions. This is particularly surprising for the vulnerabilities reported by ASI, since the remediation deadline for those libraries has already expired[4], i.e., many app developers either have not reacted to Google's reporting or did not receive a notification in the first place. The subsequent API usage analysis revealed that 91.5% of these libraries are actively used by applications, i.e., at least one API call to the library was found in the non-library code. In the remaining 8.5% of cases the library is included in the app but is not in use, i.e., it is considered dead code. This number is slightly lower than the 11.8% reported for all libraries. For the advertising library *MoPub* we were not able to find apps with one of the vulnerable library versions from the year 2015. Note that using some of these libraries is already sufficient to be vulnerable. This includes all advertisement libraries and Dropbox. There is no need to explicitly invoke specific APIs since the library's core functionality (*Dropbox* authentication or showing ads in a WebView) is triggered upon initialization without further interaction. For the remaining libraries, the vulnerable functionality has to be triggered by the application such as login at *Facebook* or certificate pinning from *OkHttp*.

Out of the 16,837 actively used libraries, 97.8% could be patched through a simple drop-in replacement of the vulnerable version with the fixed one. In 57.3% of the cases, the library could even be replaced by the most current version available. The perfect updatability result for *Airpush* is due to the fact, that the patched version is the most current version to date and includes code changes only. Therefore, all versions of the second-to-last version 8.0 could be upgraded to the latest version. *Dropbox* achieved

[4] Apps are not deleted from Google Play after the remediation phase but further app updates are rejected as long as the vulnerability remains unfixed.

the lowest auto-fix rate since there were some changes to the most frequently used APIs between 1.5.4 and the fix version 1.6.2. Note, that the actual numbers for *Airpush* and *Vungle* could even be higher since we were only able to retrieve between 1–3 versions prior to the fix version.

6.7 Discussion

In the Android app ecosystem, the majority of developers makes an increasing use of third-party libraries to enhance usability and functionality of their apps. However, those components are a double-edged sword. While alleviating development through code reuse, they have been found to be a major source of bugs and security vulnerabilities [124, 72, 136, 147]. To provide end-users reliable software, it is therefore of utmost importance to keep third-party libraries up-to-date. However, recent studies [P3, 34, 83, 101] have demonstrated that in reality, we are far from having up-to-date third-party components and as a consequence, this ultimately puts the end-user's privacy and security at risk.

Our app developer survey (see Section 6.4) was a first step towards identifying the root causes *why* developers do not update libraries. A valuable insight is that while about 60% of app developers regularly update their application (at least once per quarter), mainly for new functionality, the motivation to update the included libraries is quite low (only 33% considers updating libraries as part of the app update). Contrary to the motivation to update apps for new functionality, the main incentive to update libraries is primarily bugfixes and security fixes. However, this is impeded by the fact that 63% of all library releases mix code fixes with new content and/or non-compatible API changes (cf. Figure 6.12).

6.7.1 The Role of the Library Developer

Our survey suggests that many developers abstain from updating dependent libraries due to an expected high integration effort and to prevent incompatibilities. Our library API analysis in Section 6.5 supports this assumption. There is consistently a mismatch between expected changes, i.e., conveyed through the version number, and the actual changes based on code/API changes (the semantic version was correct only in 42% of all cases). One problem is that some of the library developers are too conservative in that they never increase the major number, e.g., *Digits* (29 versions), *FasterXML-Jackson-Core* (61 versions), or *vkontakte* (29 versions), making the three number versioning scheme an effective two number versioning scheme. Another problem is when different libraries from the same developer, e.g. Android support libraries or Google Play Service libraries, have the same release cycle and different libraries receive the same new version number independent from the actual changes. The main reason for the mismatch between expected and actual semantic version, however, is probably the wrong assessment of changes by the lib developer. This means, that the specification of the patch, minor, or major version is determined by the number of code changes and effort spent for this update rather than whether the new release is API compatible to the current version.

Another aspect is that 44% of all library updates comprise major versions. This implies that many library developers too frequently release versions that might potentially break application code. This is also backed by survey responses highlighting library update problems like *"It often impacts the rest of the code. Backwards compatibility isn't ensured and that leads to a big effort in updating the libs."* or *"Sometimes library updates break existing features, due to methods changes"*. A more careful API design and aggregating unforced changes like API renaming to fewer major versions would remedy this situation. As highlighted in Section 6.6, keeping the most frequently used APIs stable, even across major versions, also has a considerable effect on the overall updatability. In particular, library developers should spend more effort in providing dedicated releases for critical bugfixes and security fixes. In six out of ten cases (cf. Table 6.4), security fixes were even bundled as a major release, which severely impedes widespread adoption.

As there is no widely accepted library marketplace or package manager for Android, changelogs are typically the main means to communicate changes to the application developer. Since about 80% of app developers read changelogs at least from time to time, this is a good way to provide detailed information on bugfixes and API changes. However, in reality, the majority of changelog entries advertises new functionality rather than reporting (detailed) bug fixes. The fact that we could only find a single entry *security fix* illustrates the current status quo pretty well. It seems that library developers put their main focus on functionality that, according to our survey results, is only the third most important update criterion. A recent study [91] reports that API changes/removals in the Android SDK typically trigger discussions on *Stack Overflow*. A simple means of providing community support after major releases would involve an active participation of library developers in such discussions to clarify changes and provide guidelines on how to perform the upgrade.

There has also been some discussion about the usefulness of Semantic Versioning. While it is certainly not supposed to be *the* gold standard, it is, in fact, a simple and useful means for library developers to express compatibility and for consumers to quickly assess the expected library integration effort. Almost all libraries in our dataset already use the X.Y.Z scheme, however, it seems that API compatibility is not always the main factor in the versioning process. An open question remains how many developers are aware of concepts like SemVer and would be able to interpret version changes correctly. To raise the awareness, library developers could pro-actively promote SemVer compliance, e.g. by adding a *SemVer compliant* badge to their code repository. In the long term, this concept will likely become more known, at least among iOS developers, since the new *Swift* package manager enforces versioning according to SemVer rules.

6.7.2 How to Improve Library Updatability?

Based on the survey responses and the follow-up analyses there are different possibilities on how to improve library updatability for different entities of the app ecosystem. Note, that this section is giving educated advice and actionable items based on first-hand information of app developers and results aggregated from follow-up analyses on libraries

and apps from Google Play. Implementation, evaluation, and assessment of developer adaption for the proposed technical solutions are subject to future work.

The Marketplace One possibility to improve adaption is a centralized marketplace, like the Google Play Store. With the App Security Improvement program, Google introduced a service that identifies security problems in apps. It notifies the respective app developer and provides a support document on how to fix the problems. However, this service also enforces that security fixes are deployed within a reasonable time. While this helps to improve the overall application security on the market, it also comes with inherent limitations. It only warns about known vulnerabilities and app developers that are writing apps for markets other than the Play Store do not benefit. The main limitation is, however, that it only fights the symptoms and does not tackle the underlying problem of the poor library version adaption rate.

About 79% of the developers in the survey could think of a dedicated library store or package manager for Android. There are already established package managers for other ecosystems such as *nuget* (.net), *npm* (JavaScript), *Cargo* (Rust), or *Cocoapods* (iOS). There is no equivalent in size and acceptance for Android to search for libraries to date. This is also documented by our survey in which the majority of developers simply refers to *"Google"* or *"Internet"* when being asked where to search for libraries. An accepted central solution could also enforce certain library requirements or quality standards more easily. For instance, the new *Swift* package manager [17] (for macOS and soon for iOS) expects packages to be distributed as source and to be named according to SemVer rules. Source distributions might foster contributions and creation of patches through the community. This is also backed by the results of the survey in which open-source is the main criteria for library selection for 61% of developers, next to functionality with about 80%.

Development Tools In 2014 the Android Gradle plugin was introduced to give app developers a powerful dependency manager to facilitate building complex applications with a larger number of third-party components. But although this is the preferred way to integrate libraries for about 30% of app developers, there is still a high number that manually integrates libraries (20%) or uses a combination of different methods (33%). Despite Gradle's high acceptance (64% like its usability, 31% somewhat), the main criticism constitutes its poor performance and the steep learning curve that might be reasons to resort to different approaches. Google picked up this criticism and recently announced a new Gradle version for Android Studio that particularly improves build times for complex applications [11]. Another argument against mixed approaches is that including libraries manually implies a higher update effort since new versions have to be downloaded manually and there is no notification when new releases become available. This reinforces the unawareness of library updates among app developers as shown in Figure 6.9. In contrast, Android Studio 2.2 recently integrated an opt-in feature to automatically notify app developers when updates of integrated third-party libraries from remote repositories such as *Maven Central* and *JCenter* become available.

Besides improving the dependency manager, integrating our detection of library version compatibility into the IDE could be helpful to automatically classify a library update and to inform about the expected code adaption effort based on the set of used library APIs. We are currently in the process of evaluating how such a plugin could be implemented for Android Studio, the preferred IDE for about 61% of app developers in our survey.

Automated Library Updates to the Rescue? A prominent example for auto-updates is the former system component WebView that was moved to a standalone-app in Android 5.0 after a series of severe security vulnerabilities. Distributed as an app, Google can automatically push security patches to this commonly used component to reach millions of devices that do no longer receive Android OS updates. Similarly, the app update mechanism of Google Play was adapted to install app updates automatically as long as no new permissions are requested. However, patching libraries, that are part of the application bytecode, is somewhat more challenging. Although Section 6.6 has demonstrated that 85.6% of libraries could be automatically updated, in 48.2% of the cases even to the latest version, there might be additional obstacles that prevent auto-updates of minor and major releases in reality (cf. Section 6.7.3). However, limiting auto-updates to patch versions that provide critical bugfixes and security fixes, would already tremendously improve the current status quo.

One possible integration approach includes the developer specifying a subset of included libraries eligible for automatic updates. A similar approach is deployed by Google Chrome to automatically update extensions [61]. The difference, however, is that the extension developer may specify this flag. There is also no formal requirement or quality assurance required, since, in worst case, the extension could simply be disabled after an unstable update. In Android, one could further introduce an option to limit updates to patch level updates that do not introduce new functionality. According to the survey, 52% of app developers would welcome such an automated update mechanism, while only one quarter disapproves such approaches. Note, that the questionnaire asked for updates in general, not for bugfix/security fix updates in particular. Thus, the acceptance of auto-fixing critical bugs only might actually be higher.

There are also different on-device deployment strategies to integrate new library versions. One option that does not require larger modifications of the app installation routine is to integrate new libraries during on-device compilation time. In Android 6, the ahead-of-time compiler on the device compiles the entire applications' bytecode to native code. There, the compilation would have to be re-triggered for each library update, similar as for new app updates. With Android 7, the ahead-of-time compilation was replaced by a just-in-time compiler [8]. This way, library code could be updated through a forced re-compilation whenever such code is used by the app. Given the generally low library API usage, the expected compilation overhead should be negligible.

Decoupling library code from application code, i.e., moving to dynamic linking, would be another option to facilitate library updates. Dynamic linking of third-party components has been disallowed by Android and iOS for a long time due to security reasons. This changed with iOS 8, released in September 2014, when Apple addressed these security

concerns with a new kernel extension to check the integrity of app files [16], i.e., whether dynamic libraries are signed, have a valid Team Identifier and that this identifier matches the one of the containing application. Updating (compatible) libraries is then simplified to replacing the library file.

6.7.3 Threats to Validity

Programmatically determining the public API of a software component is a non-trivial task, specifically on Android where, among others, advertisement libraries are typically obfuscated with identifier renaming. We distinguish obfuscated and non-obfuscated names in a best effort approach, but for corner cases, this is generally undecidable. The same is true for whitelisting package names that are supposed to be used internally. Only the ground truth in form of a proper documentation for all library releases would provide the complete public interface. However, this information is not always available. Given that we are conservative in our filtering list, we report a lower bound on updatability, e.g., when we erroneously include an obfuscated public API which is not present in the successor version due to re-obfuscation.

We conduct our library updatability analysis based on API compatibility. This constitutes the main factor to determine whether a library can be updated without any code adaption. We do not include rare cases in which public, static class fields are renamed in subsequent library releases. While such cases can cause incompatibility, we assume that they do not occur frequently. Moreover, we do not assess whether the intended functionality is preserved in the new release, i.e., that no new bugs are introduced and no code semantics changed that cause unexpected side-effects. Changing semantics of already existing APIs is considered bad practice and strongly discouraged as there are no simple means of detecting such cases for the library consumer.

We also consider the case when libraries depend on additional libraries. Versions that include other libraries can only be updated if all sub-dependencies can be updated as well. We found that 55% of the libraries in our database include at least one version with sub-dependencies. However, through manual investigation, we identified most of these dependencies as optional. In the majority of cases, advertisement mediation frameworks can be configured to use multiple ad libraries from different providers. There are two utility libraries that are used by five other libraries, *Gson* and *okio* with an updatability of 99.9% and 100%, respectively. Hence, it is safe to assume that these sub-dependencies do not influence the updatability of libraries that include them.

Finally, we investigated changes of the minimal, required Android API level by libraries. Although this does not affect the correctness of our library updatability results, the app developer might have to increase the app's minimum API level in order to update a library. This implies that the updated application may no longer be compatible with devices having an older Android version which consequently reduces the app's potential user base. To investigate the severity of such cases we aggregated a history of changes of the minimal API for the eight libraries in Table 6.6 from publicly available changelogs. Across versions, libraries have changed the minSDK version between 1–2 times. The

most current version of five libraries requires a minimum API level between 11–16 (Android 3.0–4.1). *Dropbox* does not state this requirement explicitly, only indirectly via an Android sample (API level 19, Android 4.4). According to the latest Google Play Access Statistics [10], less than 2% of all users have a device with API level < 16 (9% with API level < 19). These results suggest that library developers are very conservative in their choice of the minimal SDK to support a wide range of devices and consequently the expected loss of potential users is negligible for app developers.

6.8 Related Work

There have been several studies on different software ecosystems to assess the ripple effect of API changes. Dig et al. [47] found that in 80% of cases API changes in libraries break the client application upon update. Kim et al. [81] investigated the relationship between library API changes and bugs. They found that the number of bugs particularly increases after API refactoring. Bavota et al. [24] studied the evolution of dependencies between Java projects of the Apache ecosystem to find that client projects are more willing to upgrade a library when the new version includes a high number of bugfixes. At the same time, API changes discouraged the user from upgrading since substantial code adaption effort might be required to include the new release. While those findings are in line with our results, i.e., mismatch of expected and actual changes and insights from app developers about why libraries are not updated, this work goes one step further. We identified root causes for this problem in the Android ecosystem. Based on our results we thoroughly discussed various options to remedy this situation that would have a high app developer acceptance (based on our survey results).

McDonnell et al. [97] studied the Android API stability and adoption and found that app developers do not quickly adopt new APIs to avoid instability and integration effort. Another study on the Android API [90] showed that including fast-changing and error-prone APIs negatively affects the app ratings in the market. In contrast to our work, these studies investigated the Android API, however, we can confirm their findings for third-party libraries as well. Particularly, for over-privileged libraries, app developers often receive negative feedback and ratings, e.g. *"some users have complained about the permissions the app requires due to libraries"* or *"Google Play services and especially maps required for some time the storage permission which led to lots of questions and negative ratings"*.

Various studies [99, 117, 70] emphasized that code reuse is widespread in the Android market and that third-party libraries account for most of it [92, 85]. At the same time, the number of critical bugs and security vulnerabilities in third-party components has steadily increased. Over the last years, they have become the weakest link and the prime attack vector of applications [124]. Dedicated research [65, 128, 29, 134] has particularly found advertisement libraries to be hazards for the end-users' security and privacy by secretly collecting private data or even opening backdoors. This has motivated a line of research to automatically detect libraries in applications [102, 40, 141, 95]. However, these early approaches were insensitive to exact library versions.

More recent studies [P3, 34] adopted Software Bertillonage techniques [41] to identify concrete library versions and to uncover that about 70% of included libs in Google Play apps are outdated by at least one version. Similar alarming results have recently been reported for other ecosystems, such as Javascript [83] and Windows [101].

As a consequence, fast response times by library developers remain ineffective and even known security vulnerabilities [132, 134, 133, 49, 62] remain a persistent threat in the app ecosystem, when app developers take on average more than 300 days to integrate the existing fixes (cf. Section 5.7.1). This follow-up work focused on finding the root causes. Ultimately, our findings allowed us to propose actionable items that are both effective in amending the library outdatedness problem and accepted by the majority of app developers.

6.9 Conclusion

With the rapidly increasing number of used libraries, large parts of Android apps consist of third-party code. Critical bugs and security vulnerabilities in such components reach a high number of end-users and put their privacy and sensitive data at risk. At the same time reality shows that important patches either reach the app consumer only after an unacceptable long period or not at all. This work is the first to identify the root causes of *why* Android app developers do not adopt new versions. Based on first-hand information of app developers and results of two empirical studies, we propose actionable items for different entities of the app ecosystem to remedy this alarming situation. We believe that tackling the underlying problem is more effective than fighting the symptoms. This approach is also preferred by Derek Weeks (vice president at Sonatype) when being asked for a long-term solution: *"It's not a story about security professionals solving the problem, it's about how we empower development with the right information about the (software) parts they are consuming."*

7

Conclusion

Due to Android's popularity and the continuing trend to store and process sensitive information on mobile devices, there is an increasing demand to first understand security risks that arise from the new app paradigm and second to assess and measure the actual state of security of available applications. This dissertation presents a line of peer-reviewed work that advances security testing of Android apps and its application framework via static code analysis techniques. In particular, we created an analysis framework that statically models the runtime behavior of apps and the Android application framework to extract privacy and security-relevant data-flows. We further propose a third-party code detection that is resilient against common code obfuscation techniques to attribute vulnerabilities and security issues to the correct developer, i.e. either the app developer or a developer of a third-party component. Applying this technique to apps from Google Play, we measured the outdatedness of these components and identified the root causes of app developers not updating third-party dependencies. Based on the gained insights we proposed actionable items to remedy the current status quo. Among others, we showed that the vast majority of third-party libraries could be automatically updated to newer versions without modifying the application code.

A particular challenge constituted the static modeling of the complex application lifecycle to approximate the runtime behavior—an imperative step towards effective and accurate security analyses. On top of these application models, we employed a slicing-based analysis to statically retarget runtime values and strings that are passed to security-relevant API calls. To overcome limitations of prior work—lengthy output traces and a high number of false positives—we added a post-processing step to statically optimize the traces with a number of semantics-preserving transformations. As a result, our approach generates smaller traces with more expressive statements that are amenable for a higher number of automatic security checks including privacy-leak detection, user-input propagation analysis, or checks on argument values for security APIs. During a manual investigation of security issues, we found that a significant number of security issues can be attributed to third-party components that are statically linked into the application. To automate vulnerability attribution we had to devise a reliable third-party library detection that works in spite of commonly used bytecode obfuscation techniques and that is capable to pinpoint exact library versions. To overcome scalability problems while keeping a high precision, we decide to ignore code instructions and to devise a detection tool LibScout that exploits code structure that is preserved in the application bytecode. LibScout enabled the automated analysis of three important security use cases for the first time: 1. Measuring the status quo of library outdatedness in applications, 2. Detecting library versions with known security vulnerabilities and 3. Attributing new vulnerabilities and security issues to the correct application component. As another benefit, we can significantly reduce the analysis time of more involved static analysis approaches, including R-Droid, by excluding detected third-party code in a pre-processing step, leaving, in the best case, the code by the application developer only. Libraries can be analyzed once in an offline step and the results can then be readily used upon detection. With LibScout we could measure that libraries in apps are, to a large extent, significantly outdated. In our follow-up study, we identified reasons for app developers not updating their code dependencies. These can be summarized as a lack of awareness and motivation and the fear of experiencing incompatibilities by

performing an update. We further identified the library developer, a lack of platform support and development tools, that assist the application developer, as contributing factors. Given the fact that such a situation can not be changed considerably within a short time frame, we investigated to which extent outdated libraries can be upgraded automatically without having to change the code of the host application. Surprisingly, the vast majority of libraries and, in particular, almost all instances of vulnerable libraries found in apps on the Google Play store are API-compatible to the patch version and could be adopted without additional code refactoring effort. Finally, we discussed short and long-term actionable items to remedy this situation. Comprehensive application security analyzes can not solely be done at application layer as major parts of Android's functionality, including access to sensitive data and operations, is provided by the application framework. Similar to the analysis of applications, reasoning about security of the app framework requires a dedicated analysis backend. However, we first had to overcome the lack of a comprehensive knowledge base that describes how to statically analyze the framework components. Parts of our contributions include how to enumerate framework functionality that is exposed to the application layer and the generation of a static runtime model that faithfully models framework-specific concurrency pattern. Building on top of this framework model, we created the first high-level classification of Android's protected resources, i.e. we identified which functionality is actually protected by Android's permission system. Leveraging our gained insights about framework internals, we were able to generate new API-to-permission mappings that excelled over prior work in terms of completeness and precision. Such permission mappings are particularly useful for many application security tests (as conducted by R-Droid) since permission-protected APIs typically either allow access to sensitive data or perform sensitive operations.

Future Research Directions The author of this dissertation envisions several directions of follow-up and future work. With R-Droid, we built a generic analysis framework for Android apps and instantiated three different security modules on top of it. Besides adding more security checks for a more comprehensive security audit, reducing the overall app processing time is a desirable goal. The increased analysis precision partially originates from the accurate data models that R-Droid produces in a pre-processing step and that consume the largest part of the overall processing time. By using LibScout to detect known libraries, we can significantly reduce the amount of code that needs to be analyzed. Another factor is the Android framework code that needs to be added to the analysis to track data-dependencies through the framework. Prior work has either tried to manually generate a lightweight, but data-dependency-preserving version of the framework to reduce code complexity or to (semi-)automatically produce expert knowledge that can be used by analyzers instead of the original framework code. However, none of these approaches achieved both an analysis framework-independent and fully automated solution. Generating a minimal version of the framework code that preserves the public interface and the data-dependencies between method input and output would thus be highly desirable.

Another promising area for follow-up work is code re-use detection. LibScout has

already been applied successfully to identify boilerplate code generated by online app generators. This allows to trace back apps to their originating generator service to measure their market share and the impact of security vulnerabilities in such boilerplate code on the overall app ecosystem [S3]. An immediate follow-up work would include an extension of LibScout to detect code-level changes. A possible approach would be to extend the profile tree with a fourth layer that includes basic blocks of the method's control-flow graph. To still be resilient against a larger number of obfuscation approaches, the basic block hash could be generated from the set of dex instruction types (e.g. call or add instruction) rather than from the actual instruction values. Another interesting direction constitutes using a locality sensitive hashing scheme like SimHash that allows to compute a distance between hashes rather than having a binary-only decision. This would further allow to determine the degree to which code has changed. Beyond more precise pinpointing of exact library versions, this technique can also be applied to detect piggy-backed apps, i.e. clones of popular apps that have been instrumented with malicious code. Prior work has discovered that such code is often triggered by a single call that has been added to known library code [86].

In Chapter 6, we assessed library updatability by analyzing API-compatibility. A yet unanswered aspect includes how often library developers change the code semantics without changing the respective API. This can potentially break application functionality at runtime, for instance, if data formats have changed. A possible approach to determine whether the result of some API changed, constitutes using a fuzzer like AFL [3] to test all library versions that implement the same API with different code. By observing and comparing the output for a set of seeds that achieves high code-coverage, one can predict with a high-probability that the code changes introduce side-effects that the library consumer has to consider during an update. To sustainably improve library up-to-dateness, implementing and testing auto-updates would be useful, for instance, through on-device compilation and an Android Studio extension to support developers (see Section 6.7).

At application framework level, we established a generic analysis framework on which many follow-up security checks can be instantiated, including an optimization of authorization hook placement and a discovery of permission check inconsistencies (see Section 4.8). Further, the author envisions a comprehensive security testing by fuzzing the public API interface of the application framework. Here, a major challenge comprises setting up the infrastructure including a reliable feedback and logging channel and a system recovery upon failures. Moreover, coping with the highly concurrent nature of the framework and the task to generate seeds for Android-specific data structures will add another layers of difficulty. Once such a framework has been established, techniques such as directed greybox fuzzing [28] could be leveraged to target interesting code locations such as native calls, i.e. calls to native libraries via the Java Native Interface that are only accessible via the application framework.

Bibliography

Author's Papers for this Thesis

[P1] Backes, M., Bugiel, S., Derr, E., Gerling, S., and Hammer, C. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS 2016)*. ACM, 2016.

[P2] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: *Proceedings of the 25th USENIX Security Symposium (SEC 2016)*. USENIX Association, 2016.

[P3] Backes, M., Bugiel, S., and Derr, E. Reliable Third-Party Library Detection in Android and its Security Applications. In: *Proceedings of the 23rd ACM Conference on Computer and Communication Security (CCS 2016)*. ACM, 2016.

[P4] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In: *Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS 2017)*. ACM, 2017.

Further Publications of the Author

[S1] Bugiel, S., Derr, E., Gerling, S., and Hammer, C. Advances in Mobile Security. In: *8th Future Security - Security Research Conference*. Fraunhofer Verlag, 2013.

[S2] Derr, E. The Impact of Third-party Code on Android App Security. *Usenix Enigma (Enigma 2018)* (2018).

[S3] Oltrogge, M., Derr, E., Stransky, C., Acar, Y., Fahl, S., Rossow, C., Pellegrino, G., Bugiel, S., and Backes, M. The Rise of the Citizen Developer: Assessing the Security Impact of Online Application Generators. In: *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P 2018)*. IEEE, 2018.

Technical Reports of the Author

[T1] Backes, M., Bugiel, S., Derr, E., and Hammer, C. *Taking Android App Vetting to the Next Level with Path-sensitive Value Analysis*. Tech. rep. A/02/2014. Center for IT-Security, Privacy and Accountability (CISPA), Saarbrücken, 2014.

References

[1] Allen, F. E. Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization*. ACM, 1970.

[2] Alpern, B., Wegman, M. N., and Zadeck, F. K. Detecting Equality of Variables in Programs. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988)*. ACM, 1988.

[3] *American Fuzzy Lop (AFL) Fuzzer*. http://lcamtuf.coredump.cx/afl/technical_details.txt. Last visited: 09/22/2017. 2017.

[4] Anderson, J. P. *Computer Security Technology Planning Study, Volume II*. Tech. rep. ESD-TR-73-51. Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, 1972.

[5] Android Developer Documentation. *Bound Services*. https://developer.android.com/guide/components/bound-services.html.

[6] Android Developer Documentation. *Shrink Your Code and Resources*. https://developer.android.com/studio/build/shrink-code.html.

[7] Android Developers. *App Security Improvement Program*. https://developer.android.com/google/play/asi.html. Last visited: 08/25/2017. 2015.

[8] Android Developers. *Android 7 for Developers*. https://developer.android.com/about/versions/nougat/android-7.0.html. Last visited: 08/25/2017. 2016.

[9] Android Developers. *App Security Improvements: Looking back at 2016*. https://android-developers.googleblog.com/2017/01/app-security-improvements-looking-back.html. Last visited: 08/25/2017. 2017.

[10] Android Developers. *Google Play Dashboard*. https://developer.android.com/about/dashboards/index.html. Last visited: 08/25/2017. 2017.

[11] Android Developers Blog. *Android Studio 3.0 Canary 1*. https://android-developers.googleblog.com/2017/05/android-studio-3-0-canary1.html. Last visited: 08/25/2017. 2017.

[12] Android Documentation: Fragment. https://developer.android.com/guide/components/fragments.html.

[13] android.com. *Android Auto*. https://www.android.com/auto/. Last visited: 09/22/2017.

[14] ANDROID.COM. *Android TV*. https://www.android.com/tv/. Last visited: 09/22/2017.

[15] ANDROID.COM. *Android Wear*. https://www.android.com/wear/. Last visited: 09/22/2017.

[16] APPERIAN. *The Impact of iOS 8 on App Wrapping*. https://www.apperian.com/mam-blog/impact-ios-8-app-wrapping. Last visited: 08/25/2017. 2014.

[17] APPLE. *Swift Package Manager Community Proposal*. https://github.com/apple/swift-package-manager/blob/master/Documentation/PackageManagerCommunityProposal.md. Last visited: 08/25/2017. 2016.

[18] ARZT, S., BODDEN, E., and RASTHOFER, S. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[19] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., TRAON, Y. le, OCTEAU, D., and MCDANIEL, P. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2014)*. ACM, 2014.

[20] AU, K. W. Y., ZHOU, Y. F., HUANG, Z., and LIE, D. PScout: Analyzing the Android Permission Specification. In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012.

[21] BACKES, M., BUGIEL, S., GERLING, S., and STYP-REKOWSKY, P. von. Android Security Framework: Extensible Multi-Layered Access Control on Android. In: *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC 2014)*. ACM, 2014.

[22] BACKES, M., BUGIEL, S., HAMMER, C., SCHRANZ, O., and STYP-REKOWSKY, P. von. Boxify: Full-fledged App Sandboxing for Stock Android. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[23] BACKES, M., GERLING, S., HAMMER, C., MAFFEI, M., and STYP-REKOWSKY, P. von. AppGuard – Enforcing User Requirements on Android Apps. In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*. Springer-Verlag, 2013.

[24] BAVOTA, G., CANFORA, G., DI PENTA, M., OLIVETO, R., and PANICHELLA, S. How the Apache Community Upgrades Dependencies: An Evolutionary Study. *Empirical Software Engineering*, 20.5 (2015).

[25] BEN GRUVER. *Dexlib Android bytecode library*. https://code.google.com/p/smali/. 2009.

[26] BINKLEY, D. Precise Executable Interprocedural Slices. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2.1-4 (1993).

[27] BLACKSHEAR, S., GENDREAU, A., and CHANG, B.-Y. E. Droidel: A General Approach to Android Framework Modeling. In: *Proceedings of the 4th ACM*

SIGPLAN Workshop on State of the Art in Program Analysis (SOAP 2015). ACM, 2015.

[28] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. Directed Greybox Fuzzing. In: *Proceedings of the 24th ACM Conference on Computer and Communication Security (CCS 2017)*. ACM, 2017.

[29] Book, T., Pridgen, A., and Wallach, D. Longitudinal Analysis of Android Ad Library Permissions. In: *Proceedings of the 2013 Mobile Security Technologies Workshop (MoST 2013)*. IEEE, 2013.

[30] Bugiel, S., Heuser, S., and Sadeghi, A.-R. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[31] Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., and Chen, Y. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society, 2015.

[32] Chaudhuri, A., Fuchs, A., and Foster, J. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. CS-TR-4991. University of Maryland, 2009.

[33] Chen, K., Liu, P., and Zhang, Y. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014.

[34] Chi, Z. M. LibDetector: Version Identification of Libraries in Android Applications. MA thesis. Rochester Institute of Technology, 2016.

[35] Chin, E., Porter Felt, A., Greenwood, K., and Wagner, D. Analyzing Inter-application Communication in Android. In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011)*. ACM, 2011.

[36] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise Analysis of String Expressions. In: *Proceedings of the 10th International Conference on Static Analysis (SAS 2003)*. Springer-Verlag, 2003.

[37] ContentProvider API documentation. `https://developer.android.com/reference/android/content/ContentProvider.html`.

[38] CrackBerry.com Forums. *WhatsApp took all my contacts and sent to their servers without asking me*. `http://forums.crackberry.com/blackberry-apps-f35/whatsapp-took-all-my-contacts-sent-their-servers-without-asking-me-649363/`. Last visited: 09/20/2017. 2011.

[39] Crussell, J., Gibler, C., and Chen, H. Attack of the Clones: Detecting Cloned Applications on Android Markets. In: *Proceedings of the 17th European*

Symposium on Research in Computer Security (ESORICS 2012). Springer-Verlag, 2012.

[40] Crussell, J., Gibler, C., and Chen, H. Andarwin: Scalable Detection of Semantically Similar Android Applications. In: *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS 2013)*. Springer-Verlag, 2013.

[41] Davies, J., German, D. M., Godfrey, M. W., and Hindle, A. Software Bertillonage: Finding the Provenance of an Entity. In: *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR 2011)*. ACM, 2011.

[42] Dean, J., Grove, D., and Chambers, C. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP 1995)*. Springer-Verlag, 1995.

[43] Developers, A. *Configure Apps with Over 64K Methods*. https://developer.android.com/studio/build/multidex.html. Last visited: 09/22/2017. 2017.

[44] Developers, A. *Here comes Treble: A modular base for Android*. https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html. Last visited: 09/22/2017. 2017.

[45] Developers, A. *Kotlin and Android*. https://developer.android.com/kotlin/index.html. Last visited: 09/22/2017. 2017.

[46] Developers, G. *Android Things*. https://developers.google.com/iot/. Last visited: 09/22/2017.

[47] Dig, D. and Johnson, R. How Do APIs Evolve? A Story of Refactoring: Research Articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18.2 (2006).

[48] Documentation: Activity. https://developer.android.com/reference/android/app/Activity.html.

[49] Dropbox Blog. *Security bug resolved in the Dropbox SDKs for Android*. https://blogs.dropbox.com/developers/2015/03/security-bug-resolved-in-the-dropbox-sdks-for-android.

[50] EC SPRIDE Secure Software Engineering Group. *DroidBench*. https://github.com/secure-software-engineering/DroidBench.

[51] Edwards, A., Jaeger, T., and Zhang, X. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In: *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS 2002)*. ACM, 2002.

[52] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013)*. ACM, 2013.

[53] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In: *Proceedings of the 20th Usenix Security Symposium (SEC 2011)*. USENIX Association, 2011.

[54] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)security. In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012.

[55] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9.3 (1987), 319–349.

[56] F-Secure Labs. *Mobile Threat Report Q1 2014*. https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf. 2014.

[57] Ganapathy, V., Jaeger, T., and Jha, S. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In: *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS 2005)*. ACM, 2005.

[58] Gartner. *Gartner Says Worldwide Sales of Smartphones Grew 7 Percent in the Fourth Quarter of 2016*. http://www.gartner.com/newsroom/id/3609817. Last visited: 09/22/2017. 2017.

[59] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST 2012)*. Springer-Verlag, 2012.

[60] Godfrey, M. W., German, D. M., Davies, J., and Hindle, A. Determining the Provenance of Software Artifacts. In: *Proceedings of the 5th International Workshop on Software Clones (IWSC 2011)*. ACM, 2011.

[61] Google. *Chrome Extensions Autoupdating*. https://developer.chrome.com/extensions/autoupdate. Last visited: 02/10/2017.

[62] Google Support. *Security Vulnerability in Vungle Android SDKs prior to 3.3.0*. https://support.google.com/faqs/answer/6313713.

[63] Gordon, M. I., Kim, D., Perkins, J. H., Gilham, L., Nguyen, N., and Rinard, M. C. Information Flow Analysis of Android Applications in DroidSafe. In: *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS 2015)*. The Internet Society, 2015.

[64] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking App Behavior Against App Descriptions. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 2014.

[65] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe Exposure Analysis of Mobile In-app Advertisements. In: *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2012)*. ACM, 2012.

[66] Graf, J. Speeding Up Context-, Object- and Field-Sensitive SDG Generation. In: *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*. IEEE, 2010.

[67] GuardSquare. *DexGuard Android Obfuscator*. https://www.guardsquare.com/dexguard.

[68] GuardSquare. *ProGuard Java Obfuscator*. http://proguard.sourceforge.net.

[69] Hammer, C. and Snelting, G. Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. *International Journal of Information Security* (2009).

[70] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In: *Proceedings of the 9th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2012)*. Springer-Verlag, 2013.

[71] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. ASM: A Programmable Interface for Extending Android Security. In: *Proceedings of the 23rd USENIX Security Symposium (SEC 2014)*. USENIX Association, 2014.

[72] Hewlett Packard Enterprise. *HPE Cyber Risk Report*. https://techbeacon.com/resources/2016-cyber-risk-report-hpe-security. Last visited: 08/25/2017. 2016.

[73] Hoffmann, J., Ussath, M., Holz, T., and Spreitzenbarth, M. Slicing Droids: Program Slicing for Smali Code. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*. ACM, 2013.

[74] Horwitz, S., Reps, T., and Binkley, D. Interprocedural Slicing Using Dependence Graphs. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*. ACM, 1988.

[75] Huang, H., Zhu, S., Chen, K., and Liu, P. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In: *Proceedings of the 22nd ACM Conference on Computer and Communication Security (CCS 2015)*. ACM, 2015.

[76] Huang, J., Li, Z., Xiao, X., Wu, Z., Lu, K., Zhang, X., and Jiang, G. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[77] IBM. *T.J. Watson Libraries for Analysis (WALA)*. http://wala.sf.net. 2006.

[78] Ishio, T., Kula, R. G., Kanda, T., German, D. M., and Inoue, K. Software Ingredients: Detection of Third-party Component Reuse in Java Software Release. In: *Proceedings of the 13th Working Conference on Mining Software Repositories (MSR 2016)*. ACM, 2016.

[79] Jeon, J., Micinski, K. K., Vaughan, J. A., Fogel, A., Reddy, N., Foster, J. S., and Millstein, T. Dr. Android and Mr. Hide: Fine-grained Security Policies on Unmodified Android. In: *Proceedings of the 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM 2012)*. ACM, 2012.

[80] Kim, J., Yoon, Y., Yi, K., and Shin, J. ScanDal: Static Analyzer for Detecting Privacy Leaks in Android Applications. In: *Proceedings of the 2012 Mobile Security Technologies Workshop (MoST 2012)*. IEEE, 2012.

[81] Kim, M., Cai, D., and Kim, S. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. ACM, 2011.

[82] Krinke, J. Slicing, Chopping, and Path Conditions with Barriers. *Software Quality Journal*, 12.4 (2004).

[83] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*. The Internet Society. 2017.

[84] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and Mcdaniel, P. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. ACM, 2015.

[85] Li, L., Bissyandé, T. F., Klein, J., and Le Traon, Y. An Investigation into the Use of Common Libraries in Android Apps. In: *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*. IEEE. 2016.

[86] Li, L., Li, D., Bissyandé, T. F., Klein, J., Le Traon, Y., Lo, D., and Cavallaro, L. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics & Security (TIFS)* (2017).

[87] Liang, T., Reynolds, A., Tinelli, C., Barrett, C., and Deters, M. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*. Springer-Verlag, 2014.

[88] Licel Corporation. *DexProtector Android Obfuscator*. `https://dexprotector.com`.

[89] Licel Corporation. *Stringer Java Obfuscator*. `https://jfxstore.com/stringer`.

[90] Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. API Change and Fault Proneness: a Threat to the Success of Android Apps. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM. 2013.

[91] Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D. How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK. In: *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. ACM, 2014.

[92] Linares-Vásquez, M., Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D. Revisiting Android Reuse Studies in the Context of Code Obfuscation

and Library Usages. In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, 2014.

[93] Liu, B., Liu, B., Jin, H., and Govindan, R. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In: *Proceedings of the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys 2015)*. ACM, 2015.

[94] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In: *Proceedings of the 19th ACM Conference on Computer and Communication Security (CCS 2012)*. ACM, 2012.

[95] Ma, Z., Wang, H., Guo, Y., and Chen, X. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016.

[96] McAfee Labs. *McAfee Mobile Security Report: Who's is watching you?* `https://www.mcafee.com/us/resources/reports/rp-mobile-security-consumer-trends.pdf`. 2014.

[97] McDonnell, T., Ray, B., and Kim, M. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In: *Proceedings of the 29th International Conference on Software Maintenance (ICSM 2013)*. IEEE, 2013.

[98] Merkle, R. C. A Digital Signature Based on a Conventional Encryption Function. In: *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO 1987)*. Springer-Verlag, 1988.

[99] Mojica, I. J., Adams, B., Nagappan, M., Dienst, S., Berger, T., and Hassan, A. E. A Large-scale Empirical Study on Software Reuse in Mobile Apps. *IEEE software*, 31.2 (2014).

[100] Nan, Y., Yang, M., Yang, Z., Zhou, S., Gu, G., and Wang, X. UIPicker: User-Input Privacy Identification in Mobile Applications. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[101] Nappa, A., Johnson, R., Bilge, L., Caballero, J., and Dumitras, T. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE. 2015.

[102] Narayanan, A., Chen, L., and Chan, C. K. Addetect: Automated Detection of Android Ad Libraries Using Semantic Analysis. In: *Proceedings of the 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2014)*. IEEE, 2014.

[103] Nauman, M., Khan, S., and Zhang, X. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2010)*. ACM, 2010.

[104] Octeau, D., Luchaup, D., Dering, M., Jha, S., and McDaniel, P. Composite Constant Propagation: Application to Android Inter-Component Communica-

tion Analysis. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*. ACM, 2015.

[105] Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., and 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 (SEC 2013)*. USENIX Association, 2013.

[106] Oltrogge, M., Acar, Y., Dechand, S., Smith, M., and Fahl, S. To Pin or Not to Pin—Helping App Developers Bullet Proof Their TLS Connections. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[107] Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In: *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*. ACM, 2009.

[108] Palm Source, Inc. *Open Binder, Version 1.0*. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/. 2005.

[109] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[110] Parse Blog. *Discovering a Major Security Hole in Facebook's Android SDK*. http://blog.parse.com/learn/engineering/discovering-a-major-security-hole-in-facebooks-android-sdk.

[111] Pearce, P., Porter Felt, A., Nunez, G., and Wagner, D. AdDroid: Privilege Separation for Applications and Advertisers in Android. In: *Proceedings of the 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2012)*. ACM, 2012.

[112] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS 2014)*. The Internet Society, 2014.

[113] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android Permissions Demystified. In: *Proceedings of the 18th ACM Conference on Computer and Communication Security (CCS 2011)*. ACM, 2011.

[114] PreEmptive Solutions. *DashO Java Obfuscator*. https://www.preemptive.com/products/dasho.

[115] Preston-Werner, T. *Semantic Versioning 2.0.0*. http://semver.org/. Last visited: 08/25/2017. 2013.

[116] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Global Value Numbers and Redundant Computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988)*. ACM, 1988.

[117] Ruiz, I. J. M., Nagappan, M., Adams, B., and Hassan, A. E. Understanding Reuse in the Android Market. In: *Proceedings of the 20th International Conference on Program Comprehension (ICPC 2012)*. IEEE. 2012.

[118] Seo, J., Kim, D., Cho, D., Kim, T., and Shin, I. FlexDroid: Enforcing In-App Privilege Separation in Android. In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[119] Shannon, D., Hajra, S., Lee, A., Zhan, D., and Khurshid, S. Abstracting Symbolic Execution with String Analysis. *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION* (2007).

[120] Shao, Y., Ott, J., Chen, Q. A., Qian, Z., and Mao, Z. M. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[121] Shekhar, S., Dietz, M., and Wallach, D. S. AdSplit: Separating Smartphone Advertising from Applications. In: *Proceedings of the 21st Usenix Security Symposium (SEC 2012)*. USENIX Association, 2012.

[122] Smardec Inc. *Allatori Java Obfuscator*. http://www.allatori.com.

[123] Son, S., Daehyeok, G., Kaist, K., and Shmatikov, V. What Mobile Ads Know about Mobile Users. In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS 2016)*. The Internet Society, 2016.

[124] Sonatype. *2016 State of the Software Supply Chain*. https://www.sonatype.com/software-supply-chain. Last visited: 08/25/2017. 2017.

[125] Song, D., Zhao, J., Burke, M. G., Sbirlea, D., Wallach, D., and Sarkar, V. Finding Tizen Security Bugs Through Whole-system Static Analysis. *CoRR*, abs/1504.05967 (2015).

[126] *Soot - Java Analysis Framework*. https://sable.github.io/soot/. 1999.

[127] Statista. *Number of available applications in the Google Play Store from December 2009 to June 2017*. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. Last visited: 09/22/2017. 2017.

[128] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating User Privacy in Android Ad Libraries. In: *Proceedings of the 2012 Mobile Security Technologies Workshop (MoST 2012)*. IEEE, 2012.

[129] Tan, L., Zhang, X., Ma, X., Xiong, W., and Zhou, Y. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In: *Proceedings of the 17th Usenix Security Symposium (SEC 2008)*. USENIX Association, 2008.

[130] Tateishi, T., Pistoia, M., and Tripp, O. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22.4 (2013).

[131] The Android Open-Source Project. *Security-Enhanced Linux in Android*. https://source.android.com/security/selinux/.

[132] The Hacker News. *Backdoor in Baidu Android SDK Puts 100 Million Devices at Risk*. https://thehackernews.com/2015/11/android-malware-backdoor.html.

[133] The Hacker News. *Facebook SDK Vulnerability Puts Millions of Smartphone Users' Accounts at Risk*. https://thehackernews.com/2014/07/facebook-sdk-vulnerability-puts.html.

[134] The Hacker News. *Warning: 18,000 Android Apps Contains Code that Spy on Your Text Messages*. https://thehackernews.com/2015/10/android-apps-steal-sms.html.

[135] Thomé, J., Shar, L. K., and Briand, L. C. Security Slicing for Auditing XML, XPath, and SQL Injection Vulnerabilities. In: *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE 2015)*. IEEE, 2015.

[136] ThreatPost. *Code reuse - A peril for secure software development*. https://threatpost.com/code-reuse-a-peril-for-secure-software-development/122476/. Last visited: 08/25/2017. 2016.

[137] Trinh, M.-T., Chu, D.-H., and Jaffar, J. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In: *Proceedings of the 21st ACM Conference on Computer and Communication Security (CCS 2014)*. ACM, 2014.

[138] Verge, T. *Android users have installed more than 65 billion apps from Google Play in the last year*. https://www.theverge.com/2016/5/18/11673942/google-users-number-2016-android-auto-wear-tv-io. Last visited: 09/22/2017. 2016.

[139] Vidas, T., Christin, N., and Cranor, L. F. Curbing Android Permission Creep. In: *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP 2011)*. IEEE, 2011.

[140] Viennot, N., Garcia, E., and Nieh, J. A Measurement Study of Google Play. In: *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014)*. ACM, 2014.

[141] Wang, H., Guo, Y., Ma, Z., and Chen, X. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, 2015.

[142] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In: *Proceedings of the 22nd Usenix Security Symposium (SEC 2013)*. USENIX Association, 2013.

[143] Watson, R., Morrison, W., Vance, C., and Feldman, B. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In: *Proceedings of the USENIX Annual Technical Conference (ATC 2003)*. USENIX Association, 2003.

[144] Wei, F., Roy, S., Ou, X., and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In: *Proceedings of the 21st ACM Conference on Computer and Communication Security (CCS 2014)*. ACM, 2014.

[145] Weiser, M. Program Slicing. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*. IEEE Press, 1981.

[146] Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., and Beznosov, K. Android Permissions Remystified: A Field Study on Contextual Integrity. In: *Proceedings of the 24th USENIX Security Symposium (SEC 2015)*. USENIX Association, 2015.

[147] Williams, J. and Dabirsiaghi, A. *The unfortunate reality of insecure libraries*. `http://www.aspectsecurity.com/research-presentations/the-unfortunate-reality-of-insecure-libraries`. Last visited: 08/25/2017. 2012.

[148] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. Linux Security Modules: General Security Support for the Linux Kernel. In: *Proceedings of the 11th Usenix Security Symposium (SEC 2002)*. USENIX Association, 2002.

[149] Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The Impact of Vendor Customizations on Android Security. In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013)*. ACM, 2013.

[150] Yang, W., Li, J., Zhang, Y., Li, Y., Shu, J., and Gu, D. APKLancet: Tumor Payload Diagnosis and Purification for Android Applications. In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communication Security (ASIACCS 2014)*. ACM, 2014.

[151] Yang, Z. and Yang, M. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In: *Proceedings of the 3rd World Congress on Software Engineering (WCSE 2012)*. IEEE, 2012.

[152] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013)*. ACM, 2013.

[153] Yu, F., Alkhalaf, M., and Bultan, T. STRANGER: An Automata-based String Analysis Tool for PHP. In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*. Springer-Verlag, 2010.

[154] Yu, F., Bultan, T., Cova, M., and Ibarra, O. H. Symbolic String Verification: An Automata-Based Approach. In: *Proceedings of the 15th International Workshop on Model Checking Software (SPIN 2008)*. Springer-Verlag, 2008.

[155] Zhang, X., Edwards, A., and Jaeger, T. Using CQUAL for Static Analysis of Authorization Hook Placement. In: *Proceedings of the 11th Usenix Security Symposium (SEC 2002)*. USENIX Association, 2002.

[156] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X. S., and Zang, B. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In: *Proceedings of the 20th ACM Conference on Computer and Communication Security (CCS 2013)*. ACM, 2013.

[157] Zheng, Y., Zhang, X., and Ganesh, V. Z3-str: A Z3-based String Solver for Web Application Analysis. In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, 2013.

[158] Zhong, N. and Michahelles, F. Where Should You Focus: Long Tail or Superstar?: An Analysis of App Adoption on the Android Market. In: *Proceedings of the 2012 SIGGRAPH Asia Symposium on Apps (SA 2012)*. ACM, 2012.

[159] Zhou, W., Wang, Z., Zhou, Y., and Jiang, X. DIVILAR: Diversifying Intermediate Language for Anti-repackaging on Android Platform. In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY 2014)*. ACM, 2014.

[160] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In: *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY 2012)*. ACM, 2012.

[161] Zhou, Y. and Jiang, X. Dissecting Android Malware: Characterization and Evolution. In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P 2012)*. IEEE, 2012.

[162] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming Information-Stealing Smartphone Applications (on Android). In: *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*. Springer-Verlag, 2011.

A

Questionnaire: Developer Survey

We asked app developers from Google Play to complete the following questionnaire. It comprises 28 questions, grouped into four blocks, i.e., professional background, app development, third-party libraries, and demographics.

Professional Background Questions

B1: Is developing Android apps your primary job?
(i) yes, (ii) no

B2: Are you developing your apps as a hobby, are you self-employed or do you work for a company? Please check all that apply.
(i) hobby, (ii) self-employed, (iii) company, (iv) other

B3: How large is your company?
(i) up to 10 employees, (ii) 10-50 employees, (iii) 50-100 employees, (iv) >100 employees

B4: How many apps have you worked on?

App Development Questions

A1: How do you develop your app/apps? (If more than one, please choose the one you use primarily)
(i) Android Studio, (ii) Eclipse, (iii) Application Generator Framework (Cordova,

Xamarin,...), (iv) other

A2: Is/Are your app/apps updated on a fixed schedule?
(i) yes, (ii) no

A3: Which intervals do you use to update your app/apps?
(i) weekly, (ii) bi-weekly, (iii) monthly, (iv) quarterly, (v) twice per year, (vi) yearly, (vii) never

A4: For which purpose do you update your app/apps? Please check all that apply.
(i) new functionality, (ii) bugfixes, (iii) library updates, (iv) other

Third-Party Library Questions

T1: Where do you search for the libraries?

T2: Do you choose libraries according to specific criteria? Please check all that apply.
(i) Popularity, (ii) Functionality, (iii) Open-Source, (iv) Closed-Source, (v) Required Permissions, (vi) Documentation, (vii) Recommendations, (viii) Ratings, (ix) Security, (x) Update frequency, (xi) other

T3: How many different library functions do your apps typically use?

T4: How do you integrate third-party libraries into your app? Please check all that apply.
(i) Add JAR file, (ii) Gradle, (iii) Ant, (iv) Maven, (v) I don't know, (vi) other

T5: Are you happy with gradle's usability?
(i) yes, (ii) somewhat, (iii) no, (iv) I don't know

T6: Can you list a few problems that you've had with gradle?

T7: Do you update the libraries in your app regularly?
(i) yes, all of them, (ii) yes, some of them, (iii) no, (iv) I don't know

T8: Why do you update your apps' libraries?
(i) New features, (ii) Bugfixes, (iii) Security fixes, (iv) I don't know, (v) other

T9: If your app were to contain outdated libraries, why would that be? Please check all that apply.
(i) Library was still working, (ii) Too much effort, (iii) Missing update documentation, (iv) Unaware of updates, (v) Prevent incompatibilities, (vi) Bad/missing library documentation, (vii) I don't care, (viii) I don't know, (ix) other

T10: Do you have positive/negative examples for libraries regarding updatability, documentation etc.? Please give details.

T11: Would you welcome automatic library updates on user devices via the Android OS in cases where they do not break functionality?
(i) yes, (ii) no, (iii) I don't mind, (iv) I don't know

T12: Which of the following do you think would help make library updates easier? Please check all that apply.
(i) Different distribution channels, (ii) Central library marketplace, (iii) Better IDE integration, (iv) System service or package manager, (v) other

T13: Have you ever encountered negative feedback/ratings solely because of included library functionality (e.g. libs that perform tracking or aggregate user data)?
(i) yes, (ii) no, (iii) I don't know

T14: What was the problem?

Demographics

D1: How old are you? Enter 0 if you don't want to answer

D2: What is your gender?
(i) male, (ii) female, (iii) I don't want to answer

D3: What is your highest educational degree?
(i) High school, (ii) College degree, (iii) Graduate degree, (iv) I don't want to answer, (v) No degree

D4: How many years of general coding experience do you have?

D5: How many years of Android experience do you have?

D6: How did you learn to write Android code? Please check all that apply.
(i) Self-taught, (ii) Class in school, (iii) Class in university, (iv) On the job, (v) Online coding course, (vi) Other