Saarland University

Department of Computer Science

Mitigating Security and Privacy Threats from Untrusted Application Components on Android

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

von
Jie Huang

Saarbrücken, 2021

Tag des Kolloquiums:	28. November 2022
Dekan:	Prof. Dr. Jürgen Steimle

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Benjamin Kaminski
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Andreas Zeller
Akademischer Mitarbeiter:	Dr. Robert Künnemann

Zusammenfassung

Aufgrund von Androids datenzentrierter und Open-Source Natur sowie von fehlerhaften/bösartigen Apps durch das lockere Marktzulassungsverfahren, ist die Privatsphäre von Benutzern besonders gefährdet.

Diese Dissertation präsentiert eine Reihe von Forschungsarbeiten, die die Bedrohung der Sicherheit/Privatsphäre durch nicht vertrauenswürdige Appkomponenten mindern. Die erste Arbeit stellt eine Compiler-basierte Kompartmentalisierungslösung vor, die Privilegientrennung nutzt, um eine starke Barriere zwischen der Host-App und Bibliothekskomponenten zu etablieren, und somit sensible Daten vor der Kompromittierung durch neugierige/bösartige Werbe-Bibliotheken schützt. Für fehleranfällige Bibliotheken von Drittanbietern implementieren wir in der zweiten Arbeit ein auf API-Kompatibilität basierendes Bibliothek-Update-Framework, das veraltete Bibliotheken durch Drop-Ins aktualisiert, um das durch Bibliotheken verursachte Zeitfenster der Verwundbarkeit zu minimieren. Die neueste Arbeit untersucht die missbräuchliche Nutzung von privilegierten Accessibility(a11y)-Funktionen in bösartigen Apps. Wir zeigen ein datenschutzfreundliches a11y-Framework, das die a11y-Logik wie eine Pipeline behandelt, die aus mehreren Modulen besteht, die in verschiedenen Sandboxen laufen. Weiterhin erzwingen wir eine Flusskontrolle über die Kommunikation zwischen den Modulen, wodurch die Angriffsfläche für den Missbrauch von a11y-APIs verringert wird, während die Vorteile von a11y erhalten bleiben.

Abstract

While Android's data-intensive and open-source nature, combined with its less-than-strict market approval process, has allowed the installation of flawed and even malicious apps, its coarse-grained security model and update bottleneck in the app ecosystem make the platform's privacy and security situation more worrying.

This dissertation introduces a line of works that mitigate privacy and security threats from untrusted app components. The first work presents a compiler-based library compartmentalization solution that utilizes privilege separation to establish a strong trustworthy boundary between the host app and untrusted lib components, thus protecting sensitive user data from being compromised by curious or malicious ad libraries. While for vulnerable third-party libraries, we then build the second work that implements an API-compatibility-based library update framework using drop-in replacements of outdated libraries to minimize the open vulnerability window caused by libraries and we perform multiple dynamic tests and case studies to investigate its feasibility. Our latest work focuses on the misusing of powerful accessibility (a11y) features in untrusted apps. We present a privacy-enhanced a11y framework that treats the a11y logic as a pipeline composed of multiple modules running in different sandboxes. We further enforce flow control over the communication between modules, thus reducing the attack surface from abusing a11y APIs while preserving the a11y benefits.

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 idea of *CompARTist* [P1] was developed together with Sven Bugiel. The systematization of integration techniques for advertisement libraries is completed by the author. The author was further responsible for the design, implementation, and performance evaluation of *CompARTist*. Oliver Schranz contributed to the robustness evaluation with his monkey-troop platform originally designed for evaluating his ARTist Modules [1]. Sven Bugiel and Oliver Schranz contributed with general writing tasks. All authors performed reviews of the paper.

Up2Crash [P2] is a follow-up work for Erik Derr's library updatability paper [2]. The author contributed to this work with the design of a two-stage updating experiment, the implementation of an automated library update framework in Stage-1, and carrying out dynamic tests on this framework in Stage-2. Nataniel Borges contributed by seeking hidden malfunctions in Stage-2 with his DroidMate tool [3]. The further root cause analysis, case studies, and library updatability re-estimation task are accomplished by the author. Sven Bugiel and Nataniel Borges were involved in general writing tasks. All authors are involved in the paper review work.

The idea of treating accessibility logic as a pipeline in accessibility feature misusing paper [P3] was initialized in a discussion with Sven Bugiel. The author was further responsible for the concrete design, implementation, and evaluation of the framework. The author took the advice from Usenix Security'20 anonymous reviewers by implementing an information flow control mechanism to the pipeline to further improve the privacy and security of the accessibility usage. Sven Bugiel contributed to general writing tasks. All authors performed reviews of the paper.

[P1] Huang, J., Schranz, O., Bugiel, S., and Backes, M. The art of app compartmentalization: compiler-based library privilege separation on stock android. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017, 1037–1049.

[P2] Huang, J., Borges, N., Bugiel, S., and Backes, M. Up-to-crash: evaluating third-party library updatability on android. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, 15–30.

[P3] Huang, J., Backes, M., and Bugiel, S. A11y and privacy don't have to be mutually exclusive: constraining accessibility service misuse on android. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

Acknowledgments

After more than five years of effort, my journey of Ph.D. study is finally approaching its end. I know that I could not have completed this dissertation without the help of others. I would like to express my most sincere gratitude to the following people.

First of all, I would like to thank my supervisor Michael Backes for providing me with such a great opportunity to be a member of the Information Security & Cryptography group. Thanks to his openness and support for research, I had the freedom to choose my topics and try new things. He also set an example for me with his sharp mind in research and relentless passion for work. It's my fortune to be his student and learn from him.

I would also like to express my special gratitude to Sven Bugiel. In the past five years, Sven has taught me a lot about how to do research and how to start an academic career. From paper writing to conference presentations, he spared no effort to guide me and provide helpful feedback. His dedication to excellence has always impressed and motivated me. I am particularly grateful for his generous help with all of my research projects. I am also thankful to other talented coauthors: Oliver Schranz and Nataniel Borges. Working with them to identify and address problems in our research was a pleasure for me.

I am also grateful to Andreas Zeller for taking the time to examine my thesis and give me valuable feedback. Thanks to Benjamin Kaminski for chairing my defense and Robert Künnemann for writing the protocol. My defense could not happen without their efforts.

Many thanks to my officemates: Erik Derr, Tin Nguyen, Oliver Schranz, Milivoj Simeonovski, Malte Skoruppa, Sebastian Weisgerber, and Yang Zou. Thank them for their kindness and help once I first joined the group. Not only do we have many productive research discussions, but we also chat about a variety of interesting topics, which creates a productive and enjoyable work environment.

Thanks also to our group members. We shared a lot of wonderful experiences. No matter it is a hallway conversation or a conference trip, they are all precious memories for me. Thanks to the staff from all departments of CISPA for helping me process documents, IT affairs, or business trips.

Last but not least, I want to thank my family for their unconditional love. Thank them for respecting my sudden decision to study far away from home. Their support and encouragement have been my best spiritual strength, accompanying me through all my challenges.

Contents

1 Introduction **1**

2 Technical Background **7**
2.1 Android Primer 8
2.1.1 Android Software Stack 8
2.1.2 Android Accessibility Framework 10
2.2 Android Security Mechanisms 12
2.3 Android Software Update Ecosystem 13

3 CompARTist **15**
3.1 Motivation 16
3.2 Problem Description 16
3.2.1 Problem of Library Blocking 16
3.2.2 Problem of Privileges Separating 17
3.3 Contributions 19
3.4 Library Integration Techniques 20
3.5 CompARTist Design 21
3.5.1 System Overview 21
3.5.2 Inter-Application Communication Channel 22
3.5.3 Compiler-based App Rewriting 26
3.5.4 Advertisement Service App 29
3.5.5 Deployment 32
3.6 Discussion 32
3.6.1 Robustness Evaluation 33
3.6.2 Performance Evaluation 34
3.6.3 Deployment Alternatives 36
3.6.4 Limitations 37
3.6.5 Future Work 38
3.7 Conclusion 39

4 Up2Crash **41**
4.1 Motivation 42
4.2 Problem Description 42
4.3 Contributions 43
4.4 Related Work 44
4.4.1 Software Patching Techniques 44

4.4.2 Android Test Input Generation Techniques . . . 46
4.5 Requirements Analysis . . . 47
4.6 Two-stage Updating Experiment . . . 48
4.6.1 Stage-1: Automated Library Update Framework . . . 48
4.6.2 Stage-2: Automated User Interface Tests . . . 55
4.7 Root Cause Analysis . . . 59
4.7.1 Findings from Monkey Testing . . . 61
4.7.2 DroidMate Finding . . . 64
4.7.3 Case Study . . . 64
4.7.4 Library Updatability Re-Estimation . . . 68
4.8 Discussion . . . 68
4.8.1 Research Sample . . . 68
4.8.2 Entangled Dependencies . . . 69
4.8.3 Framework and UI-based Testing Limitations . . . 69
4.8.4 Efforts from Multiple Parties . . . 70
4.8.5 Updating in Automated App Testing . . . 70
4.9 Conclusion . . . 70

5 Privacy-Enhanced Accessibility Framework 73
5.1 Motivation . . . 74
5.2 Problem Description . . . 74
5.3 Contributions . . . 75
5.4 Study of Accessibility Service Usage . . . 77
5.4.1 Accessibility App Sample Set . . . 77
5.4.2 Accessibility Service Configuration . . . 79
5.4.3 Accessibility API Usage . . . 80
5.4.4 Complete Accessibility Pipelines . . . 83
5.5 Key Idea and Threat Model . . . 84
5.6 Privacy-Enhanced Accessibility Framework . . . 86
5.6.1 Overview and Design Concepts . . . 86
5.6.2 Implementation . . . 88
5.7 Evaluation . . . 92
5.7.1 Case Study: TalkBack . . . 92
5.7.2 Case Study: EVA Facial Mouse . . . 93
5.7.3 Performance Overhead . . . 94
5.8 Discussion . . . 95
5.8.1 Limits and challenges . . . 95
5.8.2 Strengthening the sandbox and IFC . . . 96
5.8.3 User approval . . . 97
5.8.4 Threats to Validity . . . 97
5.8.5 Utility apps . . . 97
5.8.6 Other attacks and privacy issues . . . 98
5.9 Related Work . . . 98
5.10 Conclusion . . . 99

6 Conclusion **101**

A Appendix **115**
A.1 APIs of CompARTist's communication channel 116
A.2 Accessibility App Sample Set 116

List of Figures

2.1 BackGround: Android Software Stack 8
2.2 BackGround: Accessibility Communication Channel 10
2.3 BackGround: Example for Authorizing an AccessibilityService 11
2.4 BackGround: Android Software Update Ecosystem 14

3.1 CompARTist: CompARTist Overview 21
3.2 CompARTist: Inter-application Communication Channel 23
3.3 CompARTist: Example Protocol Run for Ad Invocations 24
3.4 CompARTist: IR of Advertisement Loading Code 29
3.5 CompARTist: Synchronization Management 30
3.6 CompARTist: Breakdown of Robustness Evaluation 34

4.1 Up2Crash: Example Scenario for API-compatibility based Updatability 47
4.2 Up2Crash: Overview of Library Update Framework 49
4.3 Up2Crash: Update Execution Environment 50
4.4 Up2Crash: Update Handler 52
4.5 Up2Crash: Monkey Evaluation Results 56
4.6 Up2Crash: Statistics for Deprecated APIs in Libraries 65
4.7 Up2Crash: Statistics for Deprecated API Usage in Apps 65
4.8 Up2Crash: Statistics for Apps Using Deprecated APIs 65
4.9 Up2Crash: Example for Entangled Dependencies 70

5.1 A11Y: Example Sandboxing for Screen Reader 86
5.2 A11Y: Example Sandboxing for Facial Access 86
5.3 A11Y: Accessibility Pipeline with Sandboxed Modules 87
5.4 A11Y: Privacy-enhanced Accessibility Framework 89
5.5 A11Y: Accessibility Pipeline for Screen Reader 94
5.6 A11Y: Accessibility Pipeline for Facial Mouse 94

List of Tables

3.1 CompARTist: Comparison of Existing Privilege Separation Approaches 18
3.2 CompARTist: Advertising Library Integration Techniques 18
3.3 CompARTist: Performance Evaluation Results 35

4.1 Up2Crash: DroidMate Analysis Results 58
4.2 Up2Crash: Categorized Exceptions Reported by Monkey Test 60
4.3 Up2Crash: Library Manifest Changes across Different Versions 66
4.4 Up2Crash: Rules to Identify Incompatible Updates 66
4.5 Up2Crash: Library Updatability Re-estimation Results 67

5.1 A11Y: Accessibility Service Configuration in Sample Apps 78
5.2 A11Y: Allowlisted Package Names in Service Configurations 78
5.3 A11Y: Patterns of Accessibility API Usage 81
5.4 A11Y: Accessibility Pipelines for Different App Types and Scenarios . . 82
5.5 A11Y: Performance Evaluation Results 95

A.1 Appendix: A11y App Sample Set (Google Play Store) 116
A.2 Appendix: Utility App Sample Set (Google Play Store) 117
A.3 Appendix: Malicious App Sample Set (Github) 118

1
Introduction

The existence of mobile devices extended the application scenarios of computational power. Handheld devices started to undertake tasks that are otherwise supposed to be processed in a traditional computing environment. The recent development of hardware and wireless communication technologies has further boosted the evolution of mobile devices. The increasing computing capacity with a relatively low cost has successfully empowered mobile devices to gain a broader reach rapidly. The latest two decades witness the global rising of the mobile industry. The number of smartphone users worldwide has surpassed 3.6 billion by the end of 2020 [4]. Among all the mobile shipments, Android is the most popular mobile operating system worldwide. To meet the needs of this large user base, the application market of Android continues to thrive today. App developers can create their applications, integrate personalized functionalities, and submit them to Google Play Store, which is Android's official app market, or alternative app markets freely. Google Play Store alone has 3.04 million available apps as of June 2020 [5].

While the growing number of Android applications enrich and facilitate daily life, they also raise serious security and privacy concerns. Firstly, since mobile devices have become closely connected to people's social life, entertainment, education, etc., various personal data such as telephone number, bank account, and medical records are available to mobile devices. The numerous amount of sensitive on-device data, combined with the huge market share make Android naturally an attractive target for attackers. Moreover, Android is an open-source platform. The openness of its internal details makes Android easier to be exploited by malicious developers. Lastly, Android has a less strict market approval process compared with other mobile platforms like iOS. App developers are legitimately to distribute their apps to end users through either marketplace, e.g., Google Play Store, or non-marketplace, like email or website, making it hard to guarantee the quality of apps. Even on Android's official Google Play Store, which has Google Play Protect [6] deployed to scan newly uploaded apps to filter malware, there is still a chance for malevolent apps to circumvent this mechanism.

Once a malevolent app is installed on a device, the existing Android security model is not sufficient to protect user data. Android provides an application-based permission management mechanism to restrict each app's data access. When a user installs an app, Android will list the app's required permissions and ask for the user's approval. This mechanism bundles all the permissions together, thus gives the user an all-or-nothing choice. Starting with Android 6.0, this install-time permission granting model is replaced with a more flexible dynamic model. Permissions are now able to be granted separately at runtime. However, the permission delegation in the updated model is still based on application boundaries, which contradicts the purpose of prohibiting untrusted or vulnerable third-party libraries from accessing user data with the host application permissions. To make the matter worse, the current Android application ecosystem involves multiple parties, resulting in an overly long update chain of libraries, and therefore timely update delivery support for third-party components is absent. In other words, even being discovered as defective, a library will remain inside the app for a long period, which leads to a large vulnerability window and exposes sensitive data to risks for quite a while. Furthermore, the design of some Android framework APIs lacks privacy protection consideration, thus is inherently privacy-flawed. For example, Android's accessibility APIs, which construct a specific accessibility data communication channel for assistive apps beyond the permission model and sandbox mechanism, could be abused effortlessly by untrusted apps for sensitive data collection.

It is noted that the current Android security model and application update ecosystem both have loopholes in preventing untrusted components from accessing user private data. Based on this fact, this dissertation introduces a line of works to mitigate privacy threats originating from untrusted application components, including optimizing the integration model for untrusted components to support dedicated privilege restriction on untrusted components, and by improving the application update ecosystem to minimize the exposure of open vulnerability window from untrusted components. With regard to restricting the privilege of untrusted components, this line of works transfers the design concept for *privilege separation*, which better manages the privilege of untrusted components by running the untrusted code in separate sandboxes, to our optimized integration model for untrusted code. In this privilege-separated integration model, the untrusted code is separated from the rest of the application code, and each untrusted compartment is sandboxed separately and can be governed by a separate set of permissions and access rights. Notably, this was applied in this dissertation to two scenarios, advertisement libraries, and accessibility framework, solving concrete challenges for each of these two settings. The key challenge of these works is that the communication channel among those now isolated compartment sandboxes should be reconstructed so that the functionality and UI of the original application can be retained. Furthermore, a mechanism to apply the least-privileged privacy policy on those isolated sandboxes is also required to achieve the goal of constraining the available privileges for untrusted components separately. This dissertation also experiments with a new library distribution approach, which breaks the library update bottleneck in the current application ecosystem, to explore the potential of reducing privacy risks by minimizing the in-app lifetime of untrusted components (here specifically vulnerable libraries).

Our work on mitigating security and privacy threats from untrusted application components on Android includes peer-reviewed publications [P1, P2, P3], which each contributed to a solution described in this dissertation. We list our contributions as follows:

CompARTist. `CompARTist` is a new privilege separation approach for mitigating privacy leakage from untrusted third-party libraries on stock Android. In a typical Android application, the host components and its third-party libraries reside in the same sandbox and share all privileges awarded to the host components by the user, putting the users' privacy at risk of intrusions by third-party libraries. Towards this problem, the existing solutions all need either system modification or app rewriting, which makes them difficult to be deployed widespread. In contrast to them, `CompARTist` partitions Android applications at compile-time into isolated, privilege-separated compartments for the host app and the included third-party libraries. A particular benefit of our approach is that it leverages compiler-based instrumentation available on stock Android versions and thus abstains from modification of the *Software Development Kit* (SDK), the app bytecode, or the device firmware. A particular challenge for separating libraries from their host apps is the reconstruction of the communication channels and the preservation of visual fidelity between the now separated host app and its libraries. We solved this challenge through new *inter-process communication* (IPC) protocols to synchronize layout and lifecycle management between different sandboxes. We demonstrated the efficiency and effectiveness of `CompARTist` by applying it to real-world apps from the Google Play Store that contain untrusted advertisement libraries.

Up2Crash. `Up2Crash` is an automatic library update framework. Apart from privilege separation for third-party libraries, updating outdated libraries timely also helps in mitigating privacy leakage from the open vulnerability window exposed by vulnerable libraries. Existing runtime apps' updatability analysis work [7] has revealed that the prevalence of outdated third-party libraries in Android apps is indeed a rampant problem and suggested that there is a great opportunity for drop-in replacements of outdated libraries, which would not even require cooperation by the app developers to update the libraries. However, all those conclusions are based on static app analysis, which can only provide an abstract view. `Up2Crash` framework is the practical extension towards this runtime apps' updatability analysis. We implemented this framework to update third-party libraries with drop-in replacements by their newer versions. Two dynamic tests are further carried out on this developer-independent update mechanism. Our results challenge the results of previous work [7] and identify impeditive factors in automatically updating libraries without involving app developers.

Privacy-enhanced Accessibility Framework. Privacy-enhanced accessibility framework is an extension of Android's original accessibility framework. In the current accessibility framework, powerful accessibility features are commonly misused by shady apps for malevolent purposes, such as stealing data from other apps. Unfortunately, existing defenses do not allow apps to protect themselves and at the same time to be fully inclusive to users with accessibility needs. To mitigate privacy leakage by misusing untrusted accessibility components while preserving the accessibility features for assistive apps, we first carried out a study to investigate how accessibility features are used in 95 existing accessibility apps of different types (malicious, utility, and a11y). Based on the results, we proposed to redesign the accessibility APIs and modeled the usage of the accessibility framework as a pipeline of code modules, which are all sandboxed separately. By policing the data flows of those modules, we achieve more fine-grained control over the access to accessibility features and the way they are used in apps, allowing a balance between accessibility functionality for dependent users and reduced privacy risks. We demonstrated the feasibility of our solution by migrating real-world apps to our privacy-enhanced accessibility framework.

Outline

We organize the remainder of this dissertation as follows. Chapter 2 introduces the technical background of the Android platform, the existing security mechanisms, and the app update ecosystem. After that, we present `CompARTist` and `Up2Crash` in Chapter 3 and Chapter 4, respectively, to describe our measures towards security and privacy leakage in third-party libraries. Chapter 5 presents our efforts in mitigating threats from the Android accessibility framework. This dissertation is concluded in Chapter 6.

2 Technical Background

In this chapter, we introduce the Android platform and its security and privacy specifics, as well as the Android software update ecosystem, to help you understand the proposed solutions in the subsequent chapters.

2.1 Android Primer

2.1.1 Android Software Stack

Android is an open-source software stack built on top of a modified Linux kernel. It was developed by the Open Handset Alliance [8], a business alliance of mobile and technology leaders led by Google, and got unveiled in 2007. Since its initial launch with commercial Android devices in 2008 [9], Android has grown and quickly dominated the mobile operating system market share. A recent report shows that Android has occupied 71.93% of the mobile OS market share in January 2021 [10]. Although primarily designed for touchscreen mobile devices like smartphones and tablets, Android and its variants are now used in a wide range of electronics such as televisions, wearables, and automobiles.

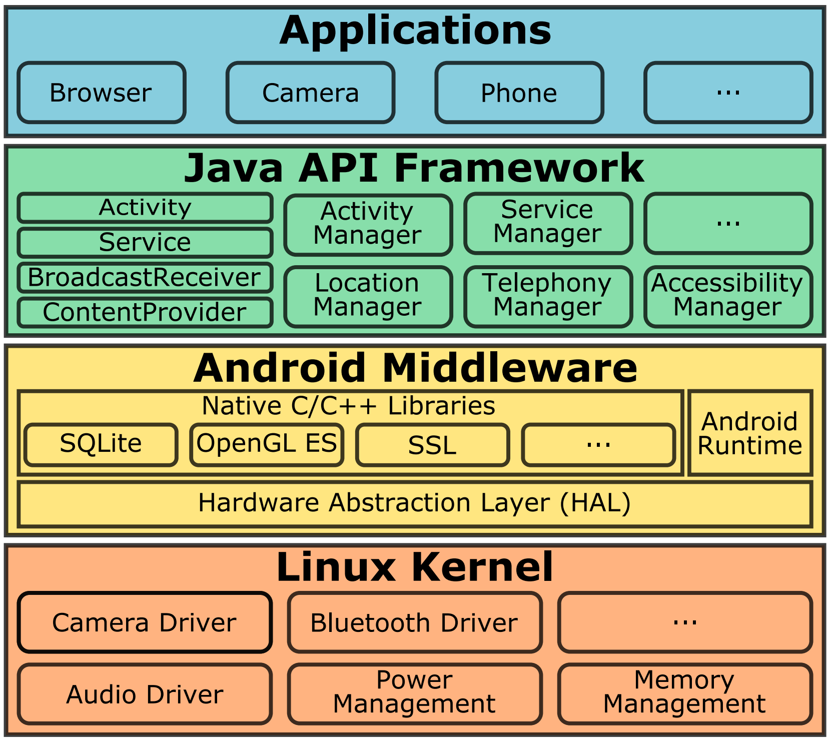


Figure 2.1: Android Software Stack

The major components of the Android software stack include a Linux kernel, a middleware infrastructure, an application framework, and various applications from bottom to top as shown in Figure 2.1. We introduce each component in the following.

Linux Kernel The bottom-most layer in Android architecture is a slightly modified version of the Linux kernel. To adapt the Linux kernel to the resource-constrained mobile embedded devices, Android made some architectural changes to the existing Linux. For example, Android includes *wakelocks* to improve power management and anonymous shared memory *ashmem* and *Low Memory Killer* to optimize memory management. It is worth noting that Android facilitates its kernel with *Binder* driver, an inter-process communication system based on OpenBinder [11], to provide flexible and reliable data exchange across process boundaries.

Android Middleware On top of the Linux kernel resides the Android middleware infrastructure which consists of a *hardware abstraction layer* (HAL), a runtime, and a native library layer. The hardware abstraction layer provides higher-level system interfaces to access hardware capabilities, such as audio, camera, and Bluetooth. It makes higher-level services and components agnostic about low-level hardware driver modifications. The runtime, together with its core libraries, is aimed at establishing the execution environment for upper-level system services and applications. Prior to Android version 5.0, the default runtime for Android was Dalvik, which is a register-based virtual machine equipped with a *just-in-time* (JIT) compiler. Start with Android version 5.0, Android implemented *Android Runtime* (ART) which utilizes *ahead-of-time* (AOT) compilation to shift the compilation expense to installation time, thus achieved better execution efficiency at runtime. Android further optimized the runtime app performance with profile-guided JIT/AOT compilation based on ART in Android version 7.0. Android also imported a set of native libraries to provide fundamental implementations like graphics, database, etc. for services and components written in C/C++.

Java API Framework Android exposes lower-level OS features to applications through Java-based APIs in the framework layer. By integrating interfaces inside the framework component `ServiceManager`, app developers can opt-in fundamental system features like location, telephony, package management, etc. to applications. Most importantly, the framework layer provides a dedicated `ActivityManagerService` service to manage the lifecycle of the app's basic components including `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. All applications have to interact with this service to launch, switch, and dispatch their components. Moreover, this framework layer also includes an accessibility framework to enable the development of assistive apps. You can find the detail in Section 2.1.2.

Applications Android developers can create both system and third-party applications by integrating framework APIs and components with the help of Android SDK. These apps offer users customized functionalities as well as high-level services exposing device resources such as telephony, internet browsing, and so on. System apps are shipped as part of the platform and maintained by platform developers, whereas third-party apps are developed and maintained by third-party developers. Before being installed on the device, each application must be signed by the developer. These signatures are utilized

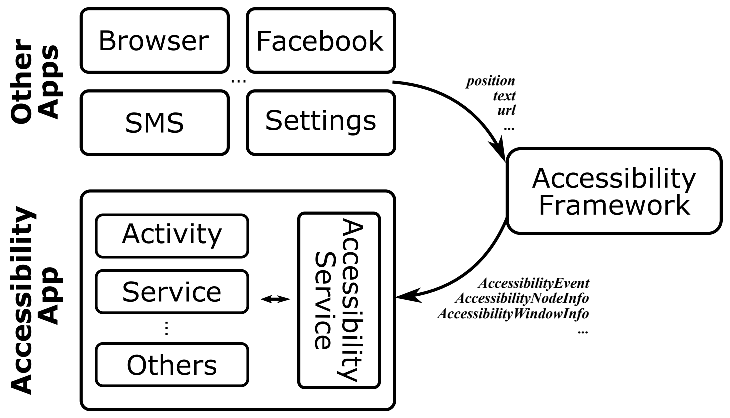
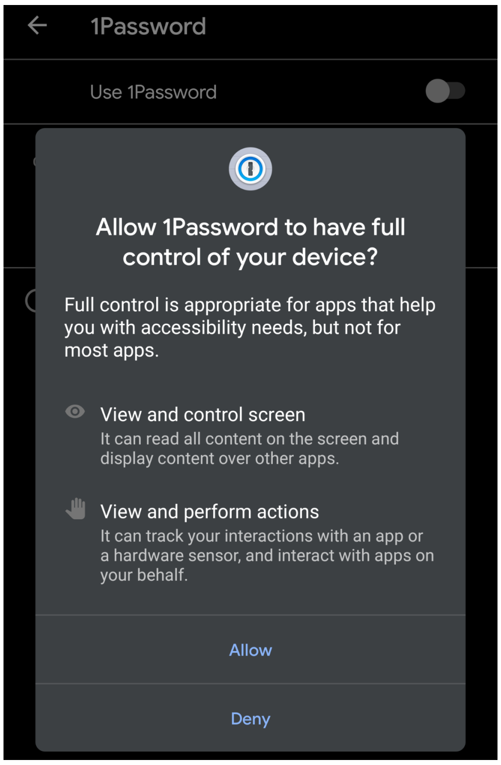


Figure 2.2: Accessibility Communication Channel

to distinguish applications and to make future updates easier to deliver. The details of Android's application update ecosystem are introduced in Section 2.3.

2.1.2 Android Accessibility Framework

We here provide technical background knowledge on Android's accessibility framework. Android supports accessibility features since API level 4 via an accessibility framework [12]. Figure 2.2 presents the overview of how this framework works. The accessibility framework acts as an intermediary between applications and accessibility apps. It monitors relevant events within applications and forwards them to accessibility apps, which in turn can use the framework to retrieve certain information (e.g., UI content) from those applications or inject events into those applications (e.g., inserting text or clicking a button). `AccessibilityService` [13] is the key component for an accessibility app to use accessibility features. Each accessibility app has to register an `AccessibilityService` to listen for accessibility events. Through `onAccessibilityEvent` callback, the app receives accessibility events that are wrapped as `AccessibilityEvent` objects. The app can then perform custom logic to consume and react to those events. For instance, a screen reader could read aloud a button description that was contained in a received event for a user's UI interaction. To register the `AccessibilityService` in the system, the developer of the accessibility app should declare it as such in the `AndroidManifest.xml` of the app. To ensure that only the system's accessibility framework can bind to this service of the app, the service declaration should require the system permission `BIND_ACCESSIBILITY_SERVICE` from any caller. Since no third-party app can successfully request this permission, the accessibility app is ensured that any caller to the `AccessibilityService` is the system. Lastly, since access to the accessibility framework is highly critical for user privacy, Android requires the user to explicitly grant this access via the *Settings* app. Figure 2.3 gives an example of this explicit activation for the *1Password* app in the system setting. Only after those steps, the accessibility app is able to assist the user (or attack them and other apps).



1Password

Use 1Password

Allow 1Password to have full control of your device?

Full control is appropriate for apps that help you with accessibility needs, but not for most apps.

View and control screen
It can read all content on the screen and display content over other apps.

View and perform actions
It can track your interactions with an app or a hardware sensor, and interact with apps on your behalf.

Allow

Deny

Figure 2.3: Example for explicitly authorizing an AccessibilityService, here of the 1Password app, in the Settings app

Accessibility Communication Channel To be able to provide assistive functionality, an `AccessibilityService` is very powerful and can access a great amount of sensitive data within other apps. Different from sensitive data that is usually protected by Android's permission model and id-based sandboxing from unauthorized access by apps, the accessibility framework can easily leak such protected data across application boundaries to an accessibility app via the accessibility communication channel. In this channel, accessibility objects, such as `AccessibilityEvent`, `AccessibilityNodeInfo`, or `AccessibilityWindowInfo`, carry other apps' sensitive data, e.g., screen text, to enable the `AccessibilityService` in doing its intended job, e.g., reading screen text aloud. An accessibility app can invoke either *global* or *node* actions. Listing 2.1 provides a toy `AccessibilityService` to illustrate this. *Global* actions are not targeting any specific app and include, for instance, invoking the device's home button or opening the recents screen (or recent task list screen) showing recently accessed apps (see Lines 6 and 8 in Listing 2.1). *Node* actions target a particular element in another app, for instance, a button or text field (see Lines 13 and 20). A typical example for this data access channel is that an accessibility app without `READ_CONTACTS` permission can still get contact information stored in the *Contacts* app through reading the text fields in `AccessibilityEvents` from the *Contacts* app.

```
public class MyAccessibilityService extends AccessibilityService {
  @Override
  public void onAccessibilityEvent(AccessibilityEvent event) {

    // global action: back to home screen
    performGlobalAction(AccessibilityService.GLOBAL_ACTION_HOME);
    // global action: show activity history
    performGlobalAction(AccessibilityService.GLOBAL_ACTION_RECENTS);

    AccessibilityNodeInfo node = event.getSource();
    if (isTargetButton(node)) {
      // local action: click the target button
      node.performAction(AccessibilityNodeInfo.ACTION_CLICK);
    } else if (isTargetEditText(node)) {
      // local action: input string "android" to an EditText
      Bundle arguments = new Bundle();
      arguments.putCharSequence(
          AccessibilityNodeInfo.ACTION_ARGUMENT_SET_TEXT_CHARSEQUENCE,
          "android");
      node.performAction(AccessibilityNodeInfo.ACTION_SET_TEXT, arguments);
    }
  }
}
```

Listing 2.1: Code example for using the accessibility service

Accessibility vs. Privacy To restrict the monitoring scope for an `AccessibilityService`, the app developer can specify an `AccessibilityServiceInfo` [14] that lists the capabilities and accessible `AccessibilityEvents` to inform the accessibility framework and the user about which events an `AccessibilityService` is interested in. Thus, the `AccessibilityServiceInfo` informs about which data is exposed to an `AccessibilityService` and what the service could do. For example, by limiting the `packageName` attribute of `AccessibilityServiceInfo` to *"com.android.settings"*, the service will only receive events for the *Settings* app. This configuration can be set either statically as meta-data inside an `XML` file or dynamically at runtime through the `setServiceInfo` interface of the accessibility framework. However, this configuration relies on the incentives of accessibility app developers. Developers of other apps can further communicate to an `AccessibilityService` that certain UI elements are not important for accessibility. This is in two different ways fallible: first, every app developer has to become active and, second, this forces app developers to choose between writing an app that is protected or that is inclusive. Moreover, this is merely an indication by the app developer and an `AccessibilityService` can decide to ignore this attribute and operate on all UI elements in a targeted app [15].

2.2 Android Security Mechanisms

Android provides different protections in different layers of the Android security model. For instance, the sandboxing mechanism in the Linux kernel, the application permission and signing mechanisms, and so on. Among all of these mechanisms, sandboxing and permission mechanisms are the most closely related to existing security and privacy threats from untrusted application components.

Application Sandbox The file system access control of Android is enforced through the sandboxing system of the underlying Linux kernel. In the original Linux, each user has a distinguish *uid* which the file access rights are assigned based on. Files generated by one user are not available to other users by default. Android transfers this *user* concept to *application*. By assigning each application with a distinct *uid*, Android runs different applications in their own process sandboxes. The data and execution status of one application are protected from being visited by other applications. In this mechanism, all app components reside in the same app sandbox and are considered as the same security principal. This kind of coarse authorization gives untrusted app components, for example, third-party libraries, the opportunity to access all files and status data from other app components, e.g., host components.

Permission Model On top of the sandbox mechanism, Android implements an application-based permission model to restrict applications' access to device resources. In this model, framework APIs exposing sensitive resources are protected by permissions. To invoke these APIs, application developers have to declare the concerned permissions in the application manifest file `AndroidManifest.xml` beforehand. There are four protection levels for Android permissions: normal, dangerous, signature, and signatureOrSystem (or signature|privileged). Among them, *dangerous* permissions are highly risky permissions that control the access to highly sensitive resources, e.g., location information or user contacts. When an app wants to invoke a *dangerous* permission-protected API, Android displays the permission request at installation time (before Android 6) or runtime (Android 6 or higher), and the app logic can only proceed once the user has confirmed the authorization. While this explicit consent mechanism works well in most cases for resource access at the application granularity level, it fails to prevent unwanted resource access from untrusted app components since all the application components share the same permission set. Furthermore, there also exists some permissions that are too powerful and coarse to protect sensitive data effectively. For example, Android provides dedicated `BIND_ACCESSIBILITY_SERVICE` permission for accessibility services. An accessibility app has to explicitly ask for the user's approval before activating any accessibility features. However, this permission is not fine-grained enough to distinguish between applications that asking for different accessibility data, and therefore violates the least-privileged privacy policy.

2.3 Android Software Update Ecosystem

We introduce the maintenance ecosystem for software on the Android platform. The official sources for Android software updates can be differentiated into four classes: app developers, Android Open Source Project (AOSP) by Google, upstream Linux kernel, and System-on-Chip (SoC) manufacturers. The updates from those sources can be delivered to end-users and take effect in their corresponding software stack layers through different update routines as shown in Figure 2.4.

The Android platform is highly diversified and fragmented. The updates from the lower layers are distributed in an arduous and time-consuming way. All the updates from AOSP, Linux kernel, and SoC manufacturers should be delivered to device manufacturers

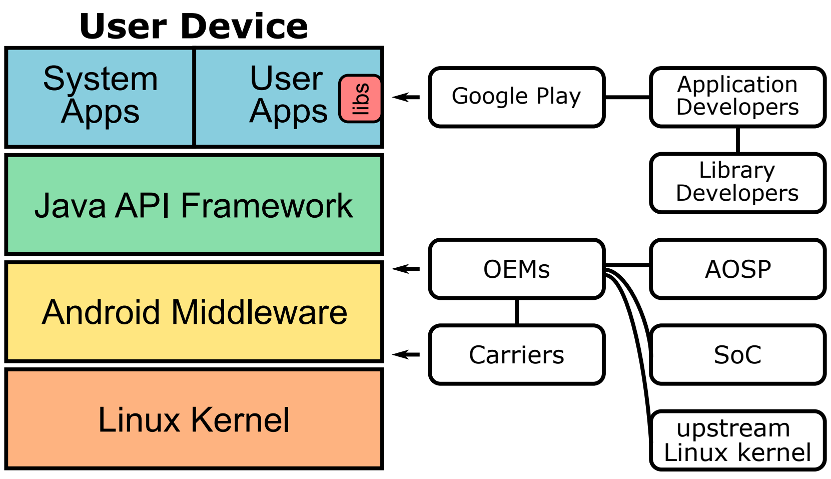


Figure 2.4: Android Software Update Ecosystem

first. After being integrated into the manufacturers' specific systems, some of those updates can be pushed to the end-users by device manufacturers, and some of them should also go through a carrier technical acceptance test at the network operator side before being delivered to users. Existing work [16] has pointed out that device manufacturers are the bottleneck in this update chain. Despite the founding of the Android Update Alliance [17], phone vendors generally lack the incentives to provide updates frequently, resulting in a long update latency or, even worse, no updates at all.

The update routine for user applications is, in contrast, pretty straightforward. The app developers submit new app versions to Google Play Store, then the target app on end devices will be reinstalled and replaced with the updated version [18]. The app updates can be delivered to user devices efficiently without any intermediate bottleneck. It is noteworthy that the updating of third-party libraries inside an application follows a similar mode to the lower layer components. New versions of third-party libraries are released by the library developers first, and after integrating the new versions into the application code by app developers, those libraries can be sent to end-users together with upgraded apps through Google Play Store. Despite the *"new library versions available"* warning provided by the built-in Lint plugin [19] of Android Studio which is the most commonly used IDE for Android app development, the integration of the updated library is highly dependent on the app developer incentives, and currently, there is no official automatic mechanism to ease this process. Google Play Store rejects apps (updates) that include libraries with known security vulnerabilities to force the developers to update those libraries through their app security improvement program[1], but this mechanism only works for a small, limited set of libraries, e.g., Apache Cordova. A lot more vulnerable libraries are still exempted from this vetting process.

[1]https://developer.android.com/google/play/asi

3 CompARTist

Compiler-based Library Privilege Separation on Stock Android

3.1 Motivation

In this chapter, we address the privacy and security issues posed by the typical untrusted application components—third-party libraries. Third-party libraries are frequently imported into applications to quicken the app development process. Compared with writing functional code from scratch, such as HTTP communication, image loading, or advertising, an existing well-encapsulated third-party package is a preferable choice for app developers. However, such external dependencies are a double-edged sword. Since third-party libraries are developed by other organizations, it is hard to guarantee their trustworthiness. As mentioned in Section 2.2, third-party libraries and their host app share the same sandbox as well as permission set on Android. This fact has been identified as a loophole for nosy libraries to exploit their ambient authority to access device-local resources, for example, location, phone identifiers, and other personal information, which can be of great interest to third parties like advertisement networks. Existing research [20, 21, 22, 23] has confirmed that those third-party libraries not only provide convenience but also bring significant risks to the users' privacy and security. In light of those risks, the security community has recently proposed various approaches to tame overly curious or even maliciously acting libraries, where the focus lies on either blocking the library functionality completely [24, 25] or privilege-separating the notorious libraries [26, 27, 28, 29, 30, 31, 32, 33].

3.2 Problem Description

The proposed library blocking solutions involve either removing the library payload or removing library code completely while library privilege-separating solutions range from dedicated system services and system modifications to application bytecode rewriting. Sadly, although these solutions greatly benefit the users' privacy and security, they do not entirely satisfy the interest of both app developers and end-users.

3.2.1 Problem of Library Blocking

Library blocking solutions are typically used in taming data-hungry advertisement libraries. The growing number of mobile advertisements has given rise to a range of approaches that follow the library blocking path in protecting user privacy and security. Some network-based advertisement filtering tools, such as AdAway [34], AdGuard [35], and AdblockPlus [36], suppress advertising functionality by either altering the device's `hosts` file or employing VPN-based content blocking. AdblockBrowser [37] integrates advertisement blocking functionality into its fully-featured browser by design to suppress advertising functionality. APKLancet [24] prunes a range of untrusted code fragments, particularly advertisement libraries, by removing the library code from the application codebase. App virtualization technology is also applied by in-app ad-blocking solutions [25] to strip advertisement from applications. Unfortunately, all these approaches can harm the free app distribution model by cutting the developers' revenue from showing advertisements.

3.2.2 Problem of Privileges Separating

Since protecting user privacy and security via library blocking inhibits the app distribution ecosystem, more solutions propose separating library privileges through library compartmentalization to eliminate privacy and security concerns while preserving the functionality of third-party libraries. We summarize these solutions from 5 aspects, including *the existence of robust privilege separation* (F1), whether *the same-origin model is preserved* (F2), whether *firmware modification or app developer effort can be exempted* (F3)(F4), and whether *privilege elevation can be avoided* (F5). We highlight their respective advantages and drawbacks in Table 3.1. According to their deployment strategies from an end user's perspective, we divided the existing library compartmentalization solutions into system-centric solutions and application-centric solutions.

System-centric Solutions Usually, system-centric solutions ship the compartmentalization approach as a part of the firmware (F3: ✗). This generally provides the possibility of constructing dedicated system services/processes for library code (F1: ✓), executing monitoring code with elevated privileges intentionally (F5: ✓), and preserving the original package and signature of the target apps (F2: ✓). Trivially, we assign the same-origin model preserving feature to all system-centric solutions since these solutions can always whitelist their own changes by customizing the signature verification process. Among the listed solutions, both AdDroid [26] and AFrame [28] include new system services that expose public APIs to applications for advertisement library integration. While these solutions achieve robust library privilege separation by executing advertisement library code in a separate process, developer effort is necessarily required to adapt their apps to the new system (F4: ✗). On the contrary, AdSplit [27] is capable of automatically retrofitting applications to use their customized system, which takes the developer out of the loop (F4: ✓). An even more involved approach, FlexDroid [29], modifies the operating system to implement inter-process stack inspection and fault isolation techniques within app processes for secure per-component permission enforcement (F1: ✗). Additionally, developers are requested to define per-component permission policies in apps' manifests (F4: ✗). The typical drawback of system-centric solutions is that system modifications are notoriously hard to distribute to end-user devices. Flashing after-market ROMs that carry library compartmentalization solutions onto devices is generally considered too high a technical hurdle for most layman end-users.

Application Layer Solutions Apart from system customization, some works take advantage of application rewriting and inlined reference monitoring (IRM [38]) techniques to abstain from firmware modification (F3: ✓). Bytecode rewriting solutions necessitate repackaging and resigning of the modified app, thus, in turn, violating Android's signature-based same-origin model. (F2: ✗; F4: ✓). Since these techniques alter the application package before installation, no higher privileges are required to operate (F5: ✓). Such rewriting and IRM techniques are widely used in privacy-enhancing

Table 3.1: Comparison of existing privilege separation approaches

Features	System-centric				Application Layer			
	FlexDroid [29]	**AdDroid** [26]	**AdSplit** [27]	**AFrame** [28]	**NativeGuard** [32]	**PEDAL** [30]	**WIREFrame** [33]	**CompARTist**
F1	✗	✓	✓	✓	✓	✗	✓	✓
F2	✓	✓	✓	✓	✗	✗	✗	✓
F3	✗	✗	✗	✗	✓	✓	✓	✓
F4	✗	✗	✓	✗	✓	✓	✓	✓
F5	✓	✓	✓	✓	✓	✓	✓	✗

F1: Robust Privilege Separation
F2: Preserves Same-Origin Model
F3: No Firmware Modification
F4: Developer Agnostic
F5: No privilege escalation/App virt.
✓: Solution provides feature
✗: Solution does not provide feature

Table 3.2: Techniques used to integrate advertising libraries with host application

Ad Lib	Share [7]	Method Invocation	Field Access	Inherit Class	Implement Interface	Custom Exception	Layout Arrangment	Android Manifest
Google Play Services Ads[†]	25.94%	✓	✓	✓	✗	✗	✓	✓
Flurry	17.85%	✓	✓	✓	✓	✗	✓	✓
Facebook Audience	12.11%	✓	✓	✓	✓	✗	✓	✓
Google Admob	9.30%	✓	✓	✗	✓	✗	✓	✓
InMobi	6.45%	✓	✓	✗	✓	✗	✓	✓
MoPub	6.13%	✓	✓	✓	✓	✓	✓	✓
Millennial Media	5.41%	✓	✓	✗	✓	✓	✓	✓
Tapjoy	4.29%	✓	✓	✗	✓	✗	✗	✓
AdColony	3.91%	✓	✓	✓	✓	✗	✓	✓
Amazon Ads	3.11%	✓	✓	✗	✓	✗	✓	✓

✓: technique used by library ✗: technique not used by library

[†] Google Play Services Ads is the successor of AdMob and comprised of several advertising networks; we only focus on the basic package that includes Banner and Interstitial ads.

solutions [39, 40, 41, 42, 43, 44] while PEDAL [30] and NativeGuard [32] are aimed specifically at privilege separation of libraries. NativeGuard moves native code libraries to a dedicated process and reconnects the libraries to the host through inter-process communication (F1: ✓). In contrast, the host and library still share the same process (F1: ✗) in PEDAL, but with API hooking implemented, the library is restricted to access sensitive resources. Lastly, the very recent WIREFrame [33], which uses app rewriting techniques for privilege-separation of not a library in particular but `WebView` components, establishes an inter-process communication-based channel between host app and remote `WebView` for remote procedure calls, lifecycle management, or restoring visual fidelity. All the application layer solutions concern repackaging and resigning, which breaks the same-origin policy of Android application updates. As a consequence, these repackaged application versions can no longer update automatically.

3.3 Contributions

To maintain the usability of library functionalities and at the same time mitigate privacy and security threats from third-party code, we propose an alternative approach to achieve privilege separation of untrusted components, in particular advertisement library, in Android apps by using *compiler*-based instrumentation of apps. Since compilation is an integrated, standardized part of app installation, compile-time modifications do not require the target application to be repackaged and resigned, hence abstaining from breaking the application signature. Moreover, Android's *dex2oat* on-device compiler can be operated entirely at the application layer and does not require changes to the application framework or system image. As such, compiler-based instrumentation forms a beneficial trade-off in the deployment of a library separation solution. The foundation of our approach to compiler-based library separation is a systematic study of the ten most frequently used advertisement libraries to identify the integration patterns between advertisement library and their host apps. We discover that only a small number of such patterns exist and that they establish only a loose coupling between libraries and host apps (e.g., callbacks, field access, or method invocations). Based on those insights, an extension for the Android on-device *dex2oat* compiler suite, which at compile-time identifies the code segments that integrate the advertisement library into the app is designed and implemented in this work. It then splits the app at those integration points into two distinct apps to be installed with a strong (process) security boundary in between and with being privileged separately. The challenge of this approach is to reintegrate the now compartmentalized library with its host app, e.g., manage the event-driven advertisement and application lifecycles or ensure visual fidelity by correctly displaying advertisements. We solve this challenge through a new IPC-based protocol for synchronizing lifecycle events between the host app's and library's sandboxes as well as for synchronizing the layout management between an overlayed advertisement and the app's user interface. More concretely, this work makes the following contributions:

Study of advertisement library integration techniques

In order to provide a solid foundation for this work, we thoroughly analyzed the ten most prevalent advertisement libraries in the Google Play Store that represent a large fraction of the market share of apps that include advertisements. Beyond motivating the design of our compartmentalization solution, we consider the results of our study to be useful for the academic audience to facilitate independent research on the topic.

Compiler-based Application Compartmentalization We introduce `CompARTist`, a compiler-based application compartmentalization system that enforces privilege separation and fault isolation of advertisement libraries on Android. Our approach offers a deployment alternative to existing solutions, since it does not require modifications of the firmware and does not break Android's signature-based same origin model. The primary challenge for our solution is the reintegration of the library compartment with the host through compile-time code instrumentations.

3.4 Library Integration Techniques

Statistical results from the freely available library detection tool LibScout [7] indicate a low fragmentation of advertising libraries among the top apps on Google Play Store. As shown in the first column in Table 3.2, between the first and the tenth most popular advertisement library, the integration rate drops down significantly from 25.94% (*Google Play Services Ads*) to 3.11% (*Amazon Ads*). In particular, this means that analyzing the ten most popular advertisement libraries allows us to cover a large fraction of all applications shipping advertisement code. Since the focus of our study is on how a host app can integrate a library, we checked the possible integration patterns by analyzing the libraries' official API documentation. For those libraries that did not provide a full list of public APIs, we use Oracle's Java class file disassembler *javap* [45] to extract the public fields and methods from the library's codebase. Table 3.2 summarizes the results of our study on possible integration techniques of advertisement libraries into host apps.

Method Invocation and *Field Access* are the two most common integration techniques among all libraries. Typically, method invocation and field access are used to exchange data between the host and the library, e.g., to request loading of an advertisement or to retrieve advertisement information.

We observed two possible techniques for deriving subclasses from library code in order to integrate the library into the app: *Class Inheritance* and *Interface Implementation.* Libraries use those techniques to allow host apps to register callback components to react to certain events, such as displaying or closing an advertisement. In many cases, the callback methods are triggered with library-specific objects as parameter values. This intertwines the library and the host tighter than, e.g., method invocations and field accesses, making the library's separation more challenging.

Furthermore, a small fraction of advertising libraries also propagate information to their hosts by throwing customized *Exceptions* that the host needs to catch and react to.

Layout Arrangement is an integration technique that allows banner advertisements to occupy part of the host app's user interface. To integrate this kind of non-full-screen views, app developers need to make changes to their apps' UI hierarchy. There are two ways to integrate a banner view element: It can either be added in the XML resource

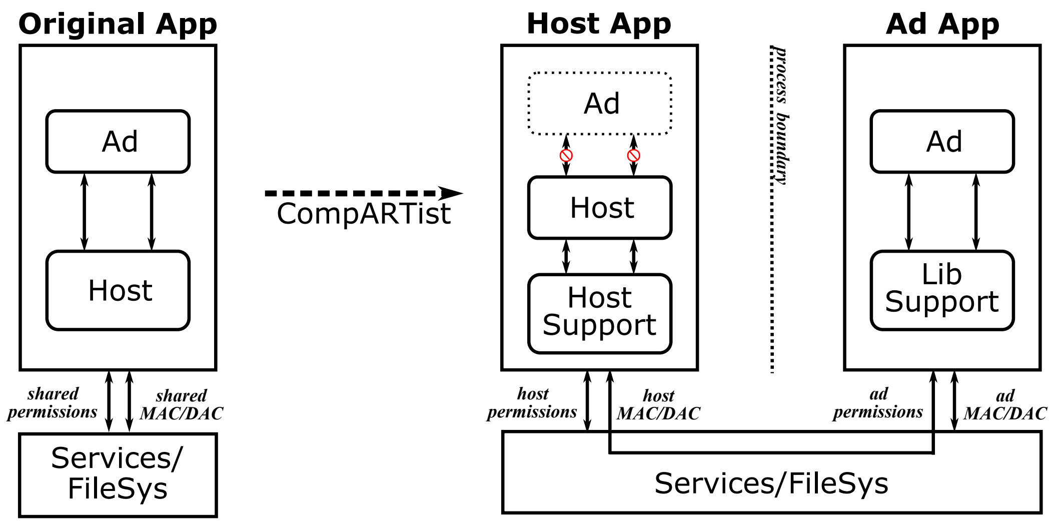


Figure 3.1: CompARTist Overview

file of its corresponding user interface or it can be instantiated and added as a new view element at runtime.

We found that all analyzed advertisement libraries require at least one permission from their host app, `INTERNET` being the most prevalent one. Further, dedicated advertisement components, e.g. `Activity`, `BroadcastReceiver`, or `ContentProvider` need to be registered for the advertisement library as well. All this requires the host app developer to make changes to the host app's *manifest file*.

Based on our findings, we conclude that most advertisement libraries share a common set of well-defined integration techniques, which makes them amenable targets for efficiently separating them at those integration points from their host apps.

3.5 CompARTist Design

We present the design and implementation of `CompARTist`.

3.5.1 System Overview

The overall design of our `CompARTist` is depicted in Figure 3.1. The goal of `CompARTist` is to privilege-separate advertisement libraries from their host apps with a strong security boundary between library and host app. Since Android's privileges are bound to *uids*, we opted in our solution for splitting an ad-supported target app into two different applications, each with a distinct *uid*. This separates advertisement libraries into a separate process with separate privileges through a distinct *uid* (F1: ✓). Since advertisement libraries are usually integrated into their host app (see Section 3.4), the primary challenge for such an approach is to re-integrated the host app and library across process boundaries. While such separation and re-integration can be achieved through firmware extensions or application rewriting (see Section 3.2.2), we present a new trade-off in the design space for Android security solutions by establishing such separation and re-integration based on an extension of the *dex2oat* on-device compiler. Operating entirely at application-level and at compile-time, this approach abstains from firmware modifications (F3: ✓), app repackaging and resigning (F2: ✓), and app developer involvement (F4: ✓) by relying solely on the ability to load the app code produced by an extended compiler backend[1] (F5: ✗).

In the remainder of this section, we explain the design and implementation of the three main components of our solution: **1)** a new IPC-based channel between host app and library that makes the previously locally integrated library remotely callable and, further, allows to synchronize the runtime states between library and app (Section 3.5.2); **2)** an extension for the *dex2oat* compiler that integrates host support for the new communication channel into the host app and replaces the library through an opaque proxy for the separated library (Section 3.5.3); and **3)** a new advertisement service app that encapsulates and privilege-separates the advertisement libraries as well as displays the ads on screen (Section 3.5.4).

3.5.2 Inter-Application Communication Channel

Since the originally app-local procedure calls to advertisement libs are not possible anymore in an isolated library design, we need an inter-application communication channel to deliver such calls remotely across process boundaries. We take advantage of the *Binder* framework [46], Android's inter-process communication mechanism, to replace the original calls to the advertisement library with remote procedure calls and transfer data, such as method parameters, between the host app and advertising service app. Figure 3.2 illustrates this channel and its components are explained in the following.

3.5.2.1 Communication Protocol and APIs

The first general challenge for our solution is the handling of data marshalling. On Android, any data that should be transferred via *Binder* IPC has to be either a primitive type (e.g., integer), String or a complex type, like a class, that implements the `Parcelable` interface to marshal the complex type into primitive types for transmission. However, library classes that were never intended to be sent via IPC, since they are only used in local invocations, do not implement this interface and are by-design not transmittable via *Binder* IPC. As a consequence, our channel cannot be used to transmit them, because it is unclear how to marshal and unmarshal those complex library classes. Thus, in `CompARTist`, we build on a generic protocol for remotely creating and operating on objects of library classes: those objects are constructed and stored at the advertisement service side and references to those objects are passed via IPC to the host app, which can use those references to invoke methods or access fields on the referenced objects. As generic, parcelable container data structure to transmit method parameters, parameter type information, and references to class instances in our

[1]`CompARTist` requires access to a particular protected directory of an app to replace the oat file that is loaded by the system. Escalated privilege, e.g., root access, is needed merely to overwrite the original oat file.

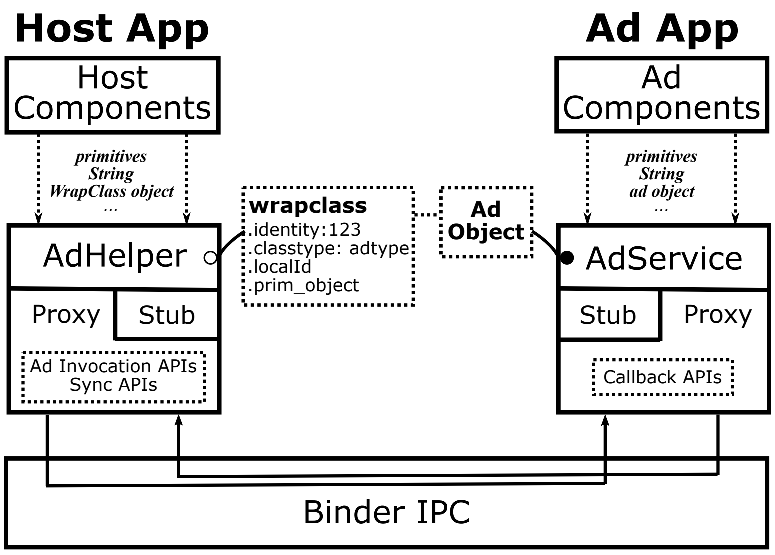


Figure 3.2: Inter-application Communication Channel

protocol, we introduce a heterogeneous key-value store with corresponding serialization and de-serialization logic called `WrapClass`.

We define three kinds of interfaces for our new inter-application communication channel that host app and advertisement service app can use to call each other via above mentioned `WrapClass`-based protocol: advertisement invocation API, callback API, and synchronization API. For each of those interface types, we automatically create `Stub` and `Proxy` classes using Android's `AIDL` [2] feature. Those classes make these communication channels more easily accessible for the host app and advertisement service app, respectively. The full interfaces for each of those interface types are listed in Appendix A.1. A particular benefit of these APIs is that they abstract from library specific methods, thus avoiding the need to generate a tailored `Stub` and `Proxy` for every available advertisement library and easing the process of adding support for new libraries.

Advertisement Invocation API Generally speaking, there are three ways for host components to communicate with the advertisement library (see also Section 3.4): instance creation, field access, and method invocation. For each of those three operations, the operation type, the operation target, and any optional parameters identify a concrete library invocation event. To better illustrate this, consider the example library invocation in Figure 3.3 where the host app creates a new `AdView` instance on which it then calls the `setAdUnitId(String)` method. First (A1 in Figure 3.3), the host requests to create an `AdView` object using the host's context. This request will be processed by `AdHelper`. `AdHelper` uses `WrapClass` to store the host's `Context` instance (A2). Since the `Context` is a non-parcelable class, `WrapClass` will only store the type information of this context parameter, i.e., class type. This `WrapClass` instance forms

[2]https://developer.android.com/guide/components/aidl.html

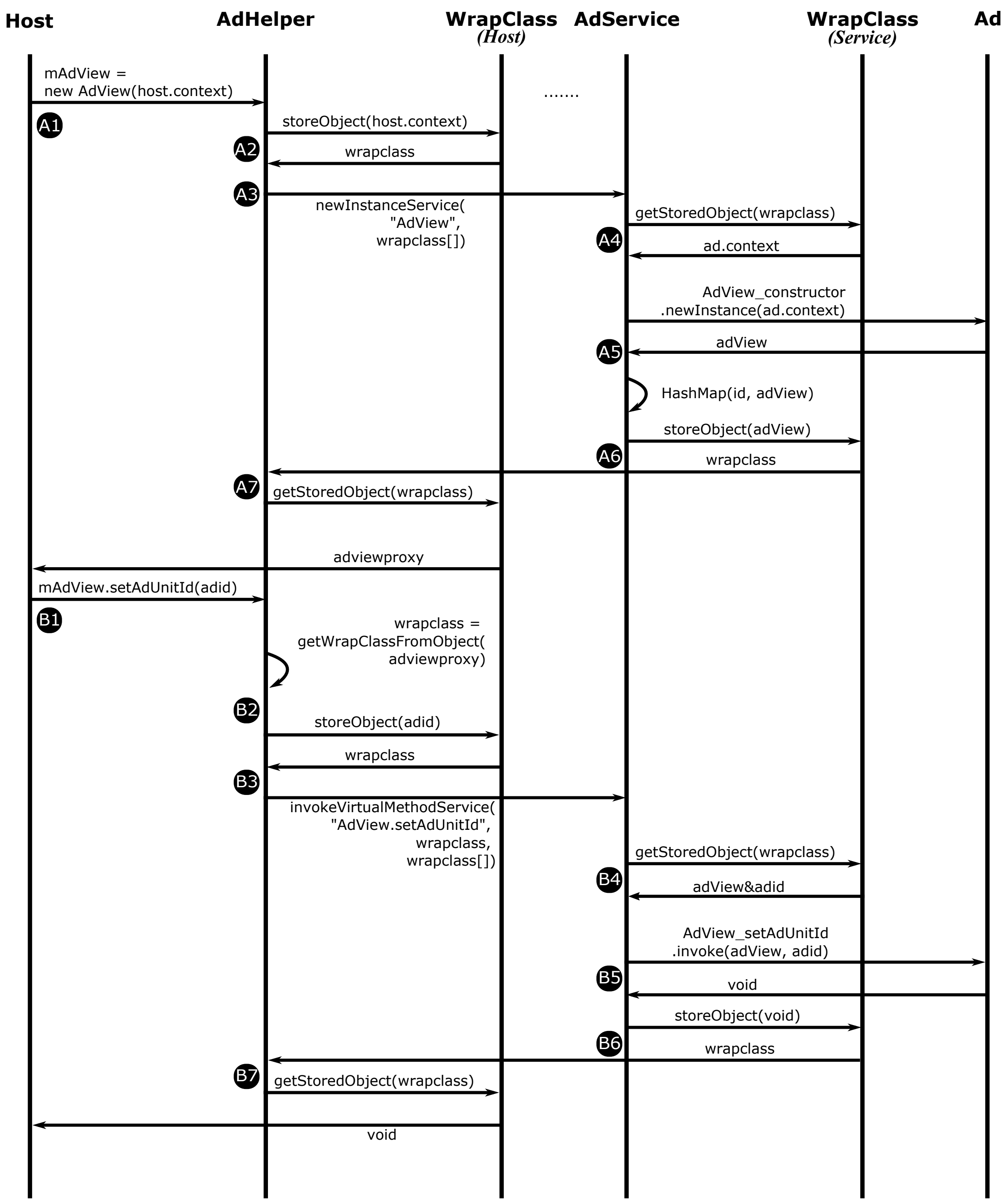

Host
AdHelper
WrapClass (Host)
AdService
WrapClass (Service)
Ad
mAdView = new AdView(host.context)
A1
storeObject(host.context)
A2
wrapclass
A3
newInstanceService("AdView", wrapclass[])
getStoredObject(wrapclass)
A4
ad.context
AdView_constructor .newInstance(ad.context)
A5
adView
HashMap(id, adView)
storeObject(adView)
A6
wrapclass
A7
getStoredObject(wrapclass)
adviewproxy
mAdView.setAdUnitId(adid)
B1
wrapclass = getWrapClassFromObject(adviewproxy)
B2
storeObject(adid)
wrapclass
B3
invokeVirtualMethodService("AdView.setAdUnitId", wrapclass, wrapclass[])
getStoredObject(wrapclass)
B4
adView&adid
AdView_setAdUnitId .invoke(adView, adid)
B5
void
storeObject(void)
B6
wrapclass
B7
getStoredObject(wrapclass)
void

Figure 3.3: Example protocol run for creating a new AdView instance and calling method setAdUnitId(String) on this instance

the container of the original context instance and together with the type information of the referenced target object (i.e., a `Context`), it is passed to the remote advertisement app through our generic IPC API as parameter of a `newInstanceSerivce` (A3) call for *"AdView"*. This API call instructs `AdService` to create a new local object with the type *"AdView"* (1st argument) and constructor parameters stored in the `WrapClass` (2nd argument). Thus, `AdService` first retrieves the stored object as the local `ad.context` parameter from the `WrapClass` object (A4). With the target class type *"AdView"* and constructor parameter, a new `AdView` object is created using the `AdService`'s context (A5). Since the channel is agnostic towards the exact library, `AdService` uses the Java reflection API to call the constructor of a class specified by the target class type parameter. This new object is stored locally in a `HashMap`, using a reference id as key. To reply to the host and return a reference to this new `AdView` instance, `AdService` stores a reference (i.e., id) together with all type information in a new `WrapClass` that it returns to the host (A6). The host creates a new proxy for this remote `AdView` object using the received type information (A7). The `WrapClass` object will also be stored in the proxy in order to establish the reference from the proxy object to the remote object.

Using such proxies, the host can invoke methods on the referenced remote objects. In Figure 3.3, the host invokes the `setAdUnitid(String adid)` method on the proxy (B1). To this end, the host stores the `adid` parameter in a `WrapClass` object and retrieves a `WrapClass` to reference the remote `AdView` object (B2). Afterwards (B3), it instructs the `AdService` to invoke the method *"setAdUnitId"* of the class *"AdView"* through the `invokeVirtualMethodService` IPC API call, where the first `WrapClass` parameter is the reference to the existing `AdView` instance on which this method should be invoked and the second `WrapClass` parameter is the argument list (i.e., wrapped `adid`). As before, `AdService` will again retrieve all parameters from the received `WrapClass` arguments (B4) and, through the reflection API, call the method on the referenced local `AdView` object (B5). It then stores the return value, here `void`, in a `WrapClass` instance (B6) and returns it to the host (B7).

Synchronization API Synchronization events only transfer meta information that indicate the supposed lifecycle state and layout of the remote advertisement. It also uses a `WrapClass`-based protocol to transfer those information, similar to invocation of advertisement libraries explained above. The purpose of this API is the continuous synchronization and smooth integration of the remote advertisement view within the `AdService` app. More details about the operations that `AdService` executes in addition to the advertisement invocations explained above are provided in Section 3.5.4.

Callback API As mentioned earlier, integrating callbacks requires a bidirectional communication flow between host and library. To solve this problem, we implement a set of callback specific APIs that the advertisement service app can use to trigger a callback method in the host app. Thus, in this case the `Proxy` is located in the service app and the `Stub` in the host app. In addition, we have to distinguish two types of callbacks: interfaces and classes. In case the callback is implemented as an extension of a library class, we additionally have to make sure that the concrete implementation's constructor is not calling its parent's constructor and hence invoking library logic in the host. Therefore, we rewrite the constructor to suppress the super call. For the interface case, this is not necessary since there is no super constructor implementation.

Otherwise, invoking callback APIs follows the same `WrapClass`-based mechanism we described earlier for the advertisement invocation API in order to invoke the callback methods of the host.

3.5.2.2 Communication Endpoints

The communication protocol is carried out between two communication entities: the host side `AdHelper` within the host app and the `AdService` in the advertisement service app, which in turn form the shim code between the host app components and our IPC channel as well as between the advertisement library components and the IPC channel, respectively (see Figure 3.2).

`AdHelper` serves as the encapsulation of our newly defined IPC APIs on the host side. `AdHelper` takes care of wrapping and unwrapping data from and to `WrapClass` and bridging the gap between our communication channel and the host components. The interfaces provided by `AdHelper` are used by our compiler-based rewriter to re-integrate the remote library into the host app by replacing local advertisement calls with calls to `AdHelper` (see following Section 3.5.3). Similar to `AdHelper` on the host side, `AdService` forms the shim between the IPC communication channel and the library's original API on the library side.

3.5.2.3 Service Connection Between Host App and Ad Service

In our current model, `AdHelper` binds itself to the `AdService` to establish the communication channel. However, this channel has to be established before any library code can be invoked by the host app in order to ensure the correct functionality of the advertising function of the host app. To solve this problem, we inject during the compilation code into the host app that scans the host app's message queue at application start to obtain the *Binder* handle of the `AdService` and then already initializes the connection to the `AdService` in a very early stage of the app's startup phase, before any `AdHelper` function is invoked, thus ensuring any library invocation finds a valid, established communication channel.

3.5.3 Compiler-based App Rewriting

In order to utilize our remote isolated advertisement library, we first need to retrofit host applications to actually use the newly introduced communication channel instead of the packaged library. Therefore, we need an application modification framework that can replace invocations to the local library with those to our remote version by redirecting all host-library interactions to the new IPC-based communication channel. Splitting the host app and local library, and afterward reintegrating the host with the IPC channel requires two essential steps: First, we need to identify the boundaries between host and advertisement code. Second, we replace all those interactions with our proxies and wrappers to restore the overall library integration across process boundaries. This results in the host app being agnostic towards the fact that it no longer interacts with the packaged advertisement library but with our remote library through an IPC channel.

3.5.3.1 Library Boundaries

The first step towards dissecting the host application is understanding the exact interaction patterns between app and library. While we discussed general integration techniques in Section 3.4, we analyzed real-world applications to identify actual code patterns with which we can transform applications properly. We distinguish between two cases: First, library objects or data are introduced into the host application by either invoking a method, accessing a field, or instantiating a class from the advertisement library. Second, library objects or data that have been introduced to the host code earlier are passed around, characterized by method return, field access, type check, or type cast within the host application. While only the first case depicts the boundary between host and library, both cases need to be considered when rewriting interactions to use our `AdHelper` instead. Apart from code boundary, special integration cases, such as manifest defined components and customized exceptions, also need specific proxy support.

3.5.3.2 Library Substitution

The second step is to utilize the information about the concrete code integration patterns to resect the library code and replace it with components from our `AdHelper`. Concretely, we utilize an app instrumentation framework that is capable of merging `AdHelper` into the application and replacing said code parts with our alternatives. In the following, we will first introduce the general structure of the host-side instrumentation part of `CompARTist` and then deep-dive into the rewriting routines as they pose one of the major challenges in establishing this new remote library connection.

3.5.3.3 ARTist Instrumentation

In this work, we leverage the Android app instrumentation capabilities of *ARTist* [1][3]. The rewriting part of *ARTist* is built on top of the *dex2oat* compiler of the *Android Runtime* (ART) introduced in Android 5 Lollipop and provides a modular framework to integrate various instrumentation solutions. We use *ARTist* to modify interactions with the advertisement library to interact with our `AdHelper` instead by utilizing two of *ARTist*'s main features: introducing customized instrumentation routines through the *module* framework and injecting our `AdHelper` into the host app through the library injection capabilities.

Module Framework *ARTist* instrumentation is based on the concept of so-called *modules*. A *module* gets full access to the application's code, allowing for arbitrary modifications, e.g., adding or removing instructions or changing them altogether, which will be reflected in the code after compilation. Internally, *ARTist* utilizes *dex2oat*'s optimization framework to disguise *modules* as optimizations and let the existing infrastructure execute them. Concretely, a *module* is then provided with the code of all methods in the compiler's internal *intermediate representation* (IR), one after another,

[3] *ARTist* is open source software available under Apache 2.0 license (https://github.com/Project-ARTist).

and can analyze and change it at will, as the compiler believes it is executing a regular optimization algorithm. As it is designed to be utilized for optimization algorithms, the compiler's IR represents a method as a control flow graph of heavily interlinked nodes that closely resemble *dex* bytecode instructions [4]. We leverage this *module* interface to implement the host side of `CompARTist`. More precisely, we introduce a specialized *module* to take care of replacing the host-library interactions with corresponding versions from our `AdHelper`.

Library Injection While the *module* framework is designed to modify existing code, the injection capability allows the merging of arbitrary own code libraries into a target application. *ARTist* will automatically make all of `AdHelper`'s APIs and other support components available to our *module* so that we can safely redirect all interactions to this new target.

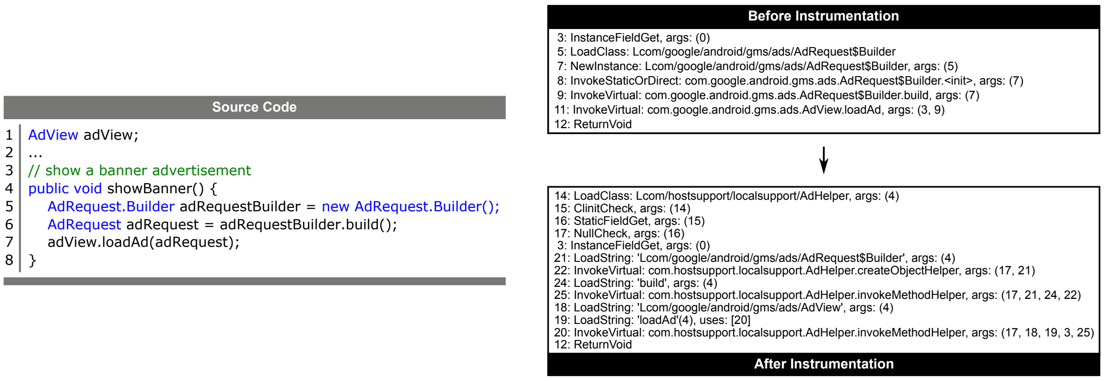
3.5.3.4 Module Design

While *ARTist* only provides the integration into the compiler, the main challenge is to design the `CompARTist` *module* to seamlessly connect the host application to the communication channel without harming the app's original semantics. Therefore, we will focus here on the design of our rewriting *module*.

Collecting Instrumentation Targets From our analysis, we know the precise patterns that bootstrap interactions between host and advertisement library. From this point, we need to find all IR code nodes that operate on the obtained library data and modify them accordingly. Since each node in the IR method graph is interlinked with its usages and inputs already, we can apply forward slicing from our starting points to find all code nodes that we need to modify. Derived from our earlier analysis, we define three types of start nodes: class loading, field access, and method invocation. As we are operating on method control flow graphs, we can find all those occurrences on a per-method base. In the IR graph, those starting points are marked by the following instructions: `LoadClass` starts a host-lib interaction by loading an advertisement library class that is subsequentially used for, e.g., `InvokeStaticOrDirect` and `NewInstance` instructions; `{Static, Instance}FieldGet` obtains previously-saved advertisement library data from a field in a host component; `InvokeVirtual` receives previously-saved advertisement library data from an invoked host method.

Instrumentation Policies Equipped with a list of entry nodes, we follow the slice through the method graph and collect every instruction that interacts with the advertisement library. Afterward, every single node is transformed to use our generic communication channel instead. This is possible since the IR graph provides us with all the structural information required to properly interact with the `AdHelper` API: operation type, operation target, and optionally, parameters. While we learn the operation type from the concrete IR node (e.g., instance creation for `NewInstance` nodes),

[4]The *ARTist* paper [1] provides in-depth documentation on the intermediate representation.



```java
AdView adView;
...
// show a banner advertisement
public void showBanner() {
    AdRequest.Builder adRequestBuilder = new AdRequest.Builder();
    AdRequest adRequest = adRequestBuilder.build();
    adView.loadAd(adRequest);
}
```

```
 3: InstanceFieldGet, args: (0)
 5: LoadClass: Lcom/google/android/gms/ads/AdRequest$Builder
 7: NewInstance: Lcom/google/android/gms/ads/AdRequest$Builder, args: (5)
 8: InvokeStaticOrDirect: com.google.android.gms.ads.AdRequest$Builder.<init>, args: (7)
 9: InvokeVirtual: com.google.android.gms.ads.AdRequest$Builder.build, args: (7)
11: InvokeVirtual: com.google.android.gms.ads.AdView.loadAd, args: (3, 9)
12: ReturnVoid
```

```
14: LoadClass: Lcom/hostsupport/localsupport/AdHelper, args: (4)
15: ClinitCheck, args: (14)
16: StaticFieldGet, args: (15)
17: NullCheck, args: (16)
 3: InstanceFieldGet, args: (0)
21: LoadString: 'Lcom/google/android/gms/ads/AdRequest$Builder', args: (4)
22: InvokeVirtual: com.hostsupport.localsupport.AdHelper.createObjectHelper, args: (17, 21)
24: LoadString: 'build', args: (4)
25: InvokeVirtual: com.hostsupport.localsupport.AdHelper.invokeMethodHelper, args: (17, 21, 24, 22)
18: LoadString: 'Lcom/google/android/gms/ads/AdView', args: (4)
19: LoadString: 'loadAd'(4), uses: [20]
20: InvokeVirtual: com.hostsupport.localsupport.AdHelper.invokeMethodHelper, args: (17, 18, 19, 3, 25)
12: ReturnVoid
```

Figure 3.4: Intermediate representation of advertisement loading code before and after the CompARTist transformation

operation target and parameters are immediately available in the graph, too, and can therefore be provided to the `AdHelper` API.

Example Transformation Figure 3.4 describes the code transformation applied to a code snippet that creates and loads a *Google Play Services Ads* advertisement. The right top part of Figure 3.4 depicts the intermediate representation of a small method that loads an `AdView`. After loading the advertisement library class (instruction 5), the result of the `LoadClass` node is used to create a new object (instruction 7 and 8). Afterward, the newly created `Builder` is used to build an `AdRequest` (instruction 9) that is consequently used to load an advertisement (instruction 11). Starting from the `LoadClass` node, forward slicing provides us with all of the above-mentioned nodes that interact with library components. The right bottom part of Figure 3.4 depicts the transformed version of the advertisement loading code. First, instead of loading and instantiating the original class, the instrumented version uses the `createObjectHelper` method from our `AdHelper` to trigger the instantiation of said object in the remote library (instruction 22). Second, the `invokeMethodHelper` allows triggering the invoked `build` method remotely (instruction 25). It only requires the name (instruction 24) and class (instruction 21) strings, and the object handle returned from `createObjectHelper` (instruction 22) to be provided as arguments. Third, the `loadAd` is remotely invoked via the `invokeMethodHelper` API (instruction 20).

3.5.4 Advertisement Service App

The advertisement service app encapsulates the advertisement library and forms the sandbox for the lib. As a separate app, executed with a distinct *uid* and in separate process, it effectively privilege-separates the advertisement library with a strong security boundary. Additionally, this app is responsible for executing operations requested by the host app on the library or for proxying callback methods from the library to the host app (as explained in Section 3.5.2). Moreover, it is responsible for displaying the advertisement on screen at the correct position to preserve visual fidelity. To correctly

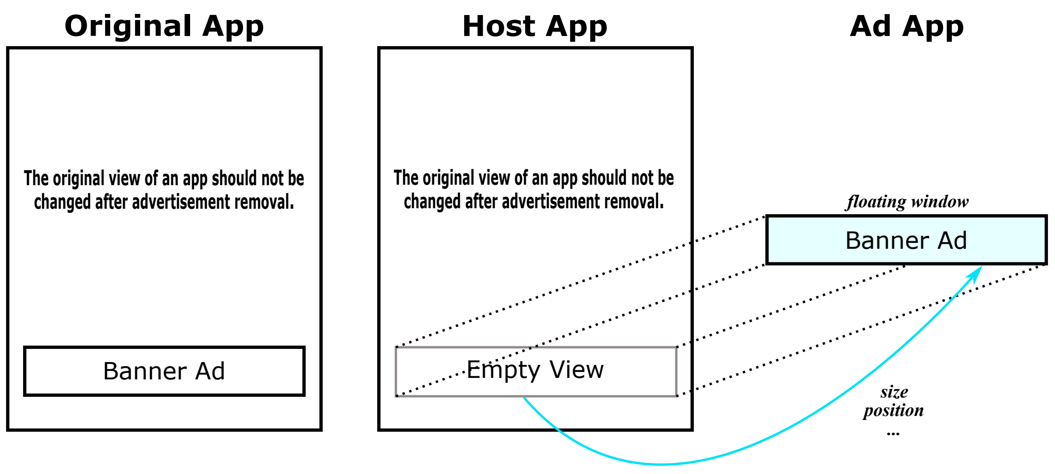

Original App
Host App
Ad App
The original view of an app should not be changed after advertisement removal.
Banner Ad
The original view of an app should not be changed after advertisement removal.
Empty View
floating window
Banner Ad
size
position
...

Figure 3.5: Synchronization Management

display ads, the `AdService` relies on lifecycle synchronization messages from the host app, e.g., show/hide an advertisement or rotate the advertisement.

3.5.4.1 Synchronizing lifecycles and preserving visual fidelity

It is important to preserve the original look-and-feel of the advertisement library (*visual fidelity*) by serving the advertisement as a part of the host application's user interface. In particular, sharing a screen with the host application is very prevalent in advertisement libraries and therefore needs careful consideration. Most advertisements are directly integrated into the layouts of their host activities and therefore share their lifecycle, such as creation, pausing, and finishing events. Thus, in `CompARTist` we need a mechanism to keep them in sync between the host app and the separately executing advertisement library in the advertisement service app.

Proxy view and floating window Instead of simply removing the original advertisement `View`, e.g., `AdView`, from the layout of the host, we replace it with a carefully crafted and empty proxy `View`. In order to preserve the dimensions and placement of the remaining GUI elements, this proxy `View` is located at the exact same position as the original advertisement `View` and occupies the exactly same space. Concurrently, advertisement service app creates a floating window that is placed on top of the proxy `View`, again occupying the very same position and space as the original advertisement `View`. It is important to note that the floating window, even though originating from the advertisement service app, can still be displayed while the host app is running in the foreground. Hence, the floating window effectively covers the same area on screen as the proxy `View` (see Figure 3.5). In our solution, we use floating window type `TYPE_TOAST` to overlay the proxy space with no additional permission needed. Whenever a lifecycle callback from the Android system arrives at the proxy `View`, such as rotation events between portrait and landscape orientation or create/pause/resume/destroy events, the proxy `View` forwards them via our inter-application communication channel and

AdService to the floating window. This allows the floating window to stay in sync with the host app's proxy View. As a result, while the advertisement is safely compartmentalized in the service app, the user perceives the advertisement as a part of the host app's layout because the occupied space and the lifecycles are synchronized. The required layout information and lifecycle events are gathered through two user interface callbacks: OnLayoutChangeListener and ActivityLifecycleCallbacks. Since the proxy View is integrated into the host layout and instantiated in the host app's context, it obtains the exact position the advertisement should have on-screen through implementation of the OnLayoutChangeListener and synchronizes this information with the remote side. By implementing ActivityLifecycleCallbacks for the proxy View, it is also straightforward to have synchronized displaying, hiding, and finishing events in the remote advertisement View.

Advertisement view inflation Usually, an advertisement view can either be defined explicitly in a layout file and inflated automatically by the system, or it can be instantiated manually at runtime. While we can handle the runtime case with our rewriting framework, supporting view replacement in case the advertisement instantiation is done by the Android framework itself is more intricate. Modifying the layout file directly is a possible solution, but it would again require to repackage the app and break the app signature. To support view substitution in both cases, at runtime and via layout files while still maintaining the app signature, we use reflection to additionally hook into the inflation mechanism at runtime and inflate our proxy View instead of the original advertisement View. Using this approach, the layout integration technique in Table 3.2 can be supported.

3.5.4.2 Multiplexing host apps

There are two approaches to achieve advertisement pairing while multiplex host apps exist. One advertisement library app per app approach, where library runs in its own remote app, can easily enforce per app privileges on the advertisement lib. This approach, however, is not resource efficient. A centralized advertisement app, which contains all advertisement libraries and serves all rewritten host apps would be more efficient. Since our inter-application communication channel between client and advertisement service app is built on top of service connections using *Binder*, the advertisement service can identify the current caller app using Binder.getCallingUid() together with information provided by the PackageManager. By using those client-specific profiles, libraries can be shared between different clients. However, this approach requires a strong domain isolation within the single user-level advertisement app to privilege advertisement executions according to their host apps (similar to AdDroid's [26] advertisement system service). Each approach has its own merits and both of them can be adopted to CompARTist, since it's just a matter of redirecting the IPC calls.

To prevent a malicious host app from stealing advertisement revenue through our CompARTist by continuously sending synchronization messages that instruct AdService to overlay *any other* app with the malicious app's advertisement, the advertisement service app must be able to make synchronization events plausible. In our current solution, we rely on the simple heuristic that only the host app that is on top of the system's `Activity` stack, i.e., in foreground on screen (excluding the floating window overlay), is able to send valid synchronization events, since it essentially instructs the `AdService` to be overlayed or finish its own overlay, thus not affecting any other app. The information about the current top `Activity` can be retrieved by third party applications (like our advertisement service app) on older Android versions via the `ActivityManager` and on newer Android versions via the `UsageStatsManager`.

3.5.5 Deployment

The key idea of our design lies in working out an application-layer-only solution that is refrained from any firmware modification while still providing robust privilege isolation. This in particular means that, as our discussion in Section 3.5.1, we tailor our solution towards enabling as many advantageous features outlined in Table 3.1 as possible. Our advertisement service app can be installed as a regular application and we here introduce the deployment of the host-side instrumentation part of `CompARTist`.

Requirements As described in Section 3.5.3, our app rewriting solution is built on top of the *ARTist* framework which is based on a modified compiler. However, to meet the above-mentioned requirements, we cannot replace the system compiler directly since not only does this concern modification of the firmware but also every single app installed afterward will be rewritten by our mechanism automatically. Instead, we want to support selective recompilation of apps, for example, the user should be able to configure the set of apps to be instrumented.

ArtistGui We fulfill this goal by utilizing *ARTistGUI* [5] which is an open-source application created for seamlessly executing *ARTist modules* from the application layer without any modification to firmware. Inside *ARTistGUI*, the compiled version of our app rewriting logic is shipped as a binary asset, and the instrumentation capabilities are exposed to users through an easy-to-use graphical interface. Once our recompilation routines are finished, the instrumentation is completely transparent to the user as the instrumented application can still start from the launcher or other apps as usual.

Dependencies While `CompARTist` abstains from modifying the Android firmware, it is still necessary to have at least elevated privileges to convince Android to run those instrumented apps rather than the original ones. We will discuss this requirement and its possible solutions as well as alternative deployment strategies for the rewriting part in Section 3.6.3.

3.6 Discussion

We evaluate the robustness of the apps transformed by `CompARTist` and the performance overhead introduced by our changes. Further, we summarize the limitation and

[5] *ARTistGUI* is open-source software available under Apache 2.0 license (https://github.com/Project-ARTist/ArtistGui).

potential improvements of our system and discuss the future research directions.

3.6.1 Robustness Evaluation

The applicability of our approach principally relies on its capability of neatly reconstructing the communication channel between the split application and advertisement code across process and sandbox border. In order to assess the robustness of our system, we carried out a large-scale evaluation of free apps from Google Play Store that contain advertisements.

3.6.1.1 Target Apps

We first collected a list of applications that integrate the *Google Play Services Ads* library as our evaluation targets. As *Google Play Services Ads* occupies a significant proportion of the mobile advertising market, evaluating with this library can reflect compatibility with a large market share of apps containing advertisements. We utilized LibScout [7], an open-source tool that can distinguish different libraries used in apps, to generate this list. Starting with top apps from Google Play Store, we filtered out apps that did not contain target library integrations or did not meet the testing prerequisites, e.g., download failed, dysfunctional behaviors (i.e., crashed at launching), or are multidex[6].

3.6.1.2 Testing Setup

To scale the evaluation of an app's dynamic status to thousands of apps, automation is the only achievable solution. We here make use of *monkey-troop*[7], a freely available testing framework designed for *ARTist modules*, to pre-filter, recompile, and explore all target apps on real devices. Specifically, after filtering out apps that do not meet our requirements, the qualified apps will be installed on the device, rewritten with `CompARTist`, and automatically launched and explored using Android's *monkey* [47] tool. We choose Google Pixel C devices running rooted stock Android 7 Nougat as our test platform and conduct the evaluation after both `CompARTist` and advertisement service app are installed and configured on these devices.

3.6.1.3 Automated UI Testing

Gaining meaningful code coverage during automated UI exploration is still an open issue. We currently take advantage of Google's *monkey* [47] tool to generate and send random touch gestures to target applications' user interfaces. However, this strategy makes the *monkey* only walk through the first few activities and all input-validated fields will be missed. In addition, though *monkey* could be constrained to access activities within a specific package, it still has the chance to leave the app UI by returning to the home screen or changing the device's quick settings randomly. Still, we use the *monkey* as our evaluation tool for the following reasons: First, the event sequences generated

[6]Our implementation does not support multidex apps (apps packaging more than one *dex* file).

[7]*monkey-troop* is open-source software available under Apache 2.0 license (https://github.com/Project-ARTist/monkey-troop).

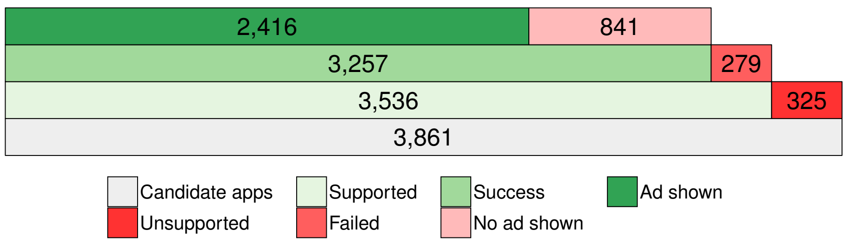


Figure 3.6: Breakdown of our robustness evaluation on applications using the Google Play Services Ads library

by *monkey* are reproducible. By re-executing *monkey* with the same seed value, the *filter* and *test* phase can share the same test setting and are prevented from exploration path mismatching. Second, code coverage is not crucial here, since there are already a lot of code instrumentations at the application start (see Section 3.5.2), and therefore launching the app to check if it crashes already suffices in most cases. So it is not a mandatory requirement to trigger at least some functionalities but a plus.

3.6.1.4 Results

We start the large-scale robustness evaluation by feeding *monkey-troop* with an initial application list of 3861 apps and show the evaluation result in Figure 3.6. After filtering out apps that do not meet the aforementioned criteria, we have 3536 qualified apps in our testing set. Among them, 3257 were tested, instrumented, and retested successfully, resulting in a success rate of 92.11% that demonstrates the compatibility between `CompARTist` and a large portion of in-app advertising applications. While the result confirms the robustness of our approach, it also gives some further information about the drawbacks of the current evaluation setting. Although we successfully instrumented 92.11% of the qualified applications, only 2416 out of 3257 (74.18%) reached an execution state that actually triggered an advertisement request during testing according to the topmost row of the result. As described in Section 3.6.1.3, the automatic *monkey* tool has limited capability in exploring advertisements that are hidden beyond the first few `Activities`. Nonetheless, please note that even for those apps that did not trigger an advertisement request, the connection between the host application and the remote advertisement service app has been successfully established, which has already involved heavy rewriting work. Hence we expect our approach to nevertheless be compatible with a significant proportion of these non-ad-triggered apps.

3.6.2 Performance Evaluation

We compare the runtime performance of the transformed app, which connects and interacts with the advertisement service app remotely with the original app so as to evaluate the performance of our approach. We focus on three representative scenarios: application start, banner advertisements, and interstitial advertisements and assess them with microbenchmarking respectively.

Table 3.3: Performance evaluation results for the app compartmentalization transformation (averaged over 50 runs)

	Baseline (ms)	**Transformed (ms)**
Application Start	6.52	149.44
Banner	2025.35	2101.50
Interstitial (Loading)	1923.05	2084.44
Interstitial (Displaying)	117.13	125.40
Interstitial (Overall)	2040.18	2209.84

3.6.2.1 Testing Setup

We again set *Google Play Services Ads* as our target library for microbenchmarking. To eliminate interference from other host functionalities and measure the immediate impact of our approach, we create a dedicated sample application that only integrates banner and interstitial advertisements according to Google's developer manual [48]. To gather the precise time consumed by certain scenarios, we embed benchmarking code into this sample application straightforwardly. By repeatedly launching the app in its original and transformed states, we can precisely catch the overhead of each scenario after applying our approach. The performance experiments are performed on a Google Nexus 6 device with rooted stock Android 7 Nougat.

3.6.2.2 Results

We exhibit the test result of performance overhead for different scenarios in Table 3.3.

Application Start As described in Section 3.5.2, the transformed application needs to establish the connection to the remote advertisement service app once upon launching, hence all its local operations are blocked until it gets the service handle. More precisely, we set a fixed amount of waiting time, which is 100 ms in our setting, before inspecting the message queue so as to ensure the availability of the advertisement service *Binder*[8]. Apart from that, it also takes the host side some time to accomplish the client-specific initialization works before entering the app's original logic, hence blocks the app further. Table 3.3 depicts the combined one-time cost covering all of the above overheads.

Banner Advertisement Considering that loading and displaying banner advertisements is a synchronous task, we start the microbenchmark when the banner is requested and terminate the test as soon as the successful loading callback is invoked. According to Table 3.3, our approach introduces an acceptable overhead of 3.62% for banner advertisement.

Interstitial Advertisement Compared to banners, interstitials are larger in size, so we follow Google's official best practice [49] of pre-loading interstitials as early as possible

[8]https://developer.android.com/reference/android/os/Binder

to ensure that it is fully loaded when display time comes. The implications of this best practice are twofold: First, only the interstitial loading concerns network traffic. Once the advertisement is fully loaded, the display of the interstitial is completely independent of the network, so its results are more reliable and easier to replicate. Secondly, if most app developers follow this best practice when integrating *Google Play Services Ads*, the interstitials will be loaded earlier asynchronously, so the user experience will not be hampered by ad-loading-induced deviations. Given those implications, we divide the performance evaluation for interstitials into a loading phase and a display phase, and set up separate microbenchmarks for these two phases. As shown in Table 3.3, the loading phase dominates the overall interstitial time consumption. Since this phase is significantly impacted by the involved network communication, we take the captured overhead of 7.74 % with a pinch of salt. However, at the very least, the measurements demonstrate that our approach has no significant impact on the loading performance. For the interstitial display phase, as expected, we notice that our approach introduces a small overhead of 6.59 % due to the additional computations and IPC roundtrips, which is still within the range of being almost imperceptible to the users.

3.6.3 Deployment Alternatives

We here discuss the existing deployment alternatives, including their drawbacks and the particular use cases that motivated them.

3.6.3.1 Host-side Deployment Alternatives

`CompARTist` is capable of being retrofitted to achieve a different subset of the objectives listed in Table 3.1 based on the concrete use cases. We present potential host-side deployment alternatives that substitute or combine the host-side modification of our approach with existing solutions.

Instrumentation Frameworks Most of the long-established Android application instrumentation frameworks are based on bytecode rewriting techniques. `CompARTist` inherits *ARTist*'s instrumentation capabilities which are on par with the existing works. Hence, for cases that application signature preservation is not necessary, our host-side rewriting can be replaced with one of the existing instrumentation frameworks [39, 40, 41, 42, 43, 44] without affecting the establishment of the new communication channel between the host app and the remote library.

Virtualization Techniques File system virtualization techniques are used by existing virtualization solutions [50, 51] to manage the interaction between applications and middleware or kernel. They are operating at application granularity—every target application is treated as a black box, thus, virtualization solutions themselves are not sufficient in coping with application retrofitting works involved in our scenario, in which case the ability to modify at instruction granularity is required so that the host and library code can be clearly distinguished and our `AdHelper` can be applied to reconstruct the communication channel. Nevertheless, we can combine virtualization solutions such as Boxify [50] or NJAS [51] either directly with *ARTist* following the suggestion in the *ARTist* paper [1], or with one of the above-mentioned instrumentation frameworks to avoid the privilege elevation required in our approach.

3.6.3.2 System-centric Deployment Alternative

We then discuss the custom ROM cases where the compartmentalization functionalities can be fully integrated into the firmware by design. In these cases, objectives like application-layer-only deployment and preserving app signatures are no longer necessary, thus it is sufficient to replace Android's default *dex2oat* compiler with an *ARTist* version running our `CompARTist` *module* straightforwardly, as this customized compiler will instrument each application automatically to replace the local advertisement calls with remote advertisement service calls. This deployment strategy can be applied to security-focused ROMs where a hardened OS version is delivered.

3.6.4 Limitations

We explain the shortcomings of our approach as well as those inherited from our prototype implementation.

Inherent Limitations As a result of the design decisions made during our system build process, we understand the existence of some inherent drawbacks in our selected approach. Whilst we establish `CompARTist` with the idea that arbitrary Android libraries can be compartmentalized in mind, it might be too aggressive to target our approach at isolating more tightly-integrated and deeply-coupled libraries like Guava [52]. As opposed to advertisement libraries that have well-defined interfaces and interact rarely with the host app, more class proxying efforts are required to reconstruct the IPC channel in isolating deeply-coupled libraries, which significantly increases the performance overhead and impairs user experience. While a novel and robust approach to compartment libraries in general is presented in this work, it is more suitable for our approach to isolate loosely-coupled libraries like advertisement libraries. Another problem of `CompARTist` is that it distinguishes advertisement libraries through a whitelist, hence it cannot deal with new libraries beyond its knowledge. Even though our current design enables the quick construction of the required remote advertisement library package, explicit human efforts are still required to extend support for additional libraries. Despite the advertisement market is not heavily fragmented at the time of this work, there is still the possibility that some new libraries might emerge in the future. One potential solution towards this could be sharing the above-mentioned library supports through, e.g., the community.

Inherited Limitations Besides limitations in our approach, `CompARTist` also inherits some inherent shortcomings of the utilized *ARTist* system. Android imported *dex2oat* since Android 5 Lollipop while *ARTist* makes use of the *Optimizing* backend from Android 6 Marshmallow. Only later Android versions can be used for the deployment of *ARTist*, which inevitably reduces the applicability of our approach. In this work, we built `CompARTist` on top of the Android 7 version of *ARTist*. Furthermore, *ARTist* has the requirement for elevated privileges, e.g., root. Fortunately, as previously described, alternative deployment strategies based on use cases exist, which allows for more relaxed requirements. Finally, as already mentioned in the robustness evaluation, our approach lacks support for multidex applications. Therefore, larger applications cannot be transformed with our current prototype if proper multidex supports are missing.

3.6.5 Future Work

We here present potential improvements to our prototype as well as future research directions.

3.6.5.1 Improvements

We can improve our work by addressing issues from both `CompARTist` and the evaluation pipeline.

Obfuscation Support To replace the original advertisement calls with calls to `Ad-Helper` in our solution, the *ARTist module* traverses the target application's code to locate these intra-application invocations to the target library. Indeed obfuscation can hide the real interfaces (that are available from the library documentation), but the library structural information has a great opportunity to be preserved. For example, LibScout [53] generates distinguishable features for different libraries based on package structure information, allowing its library detection functionality to survive code-based obfuscation. Hence, the robustness of advertisement call detection in our approach can be optimized if these library detection techniques are imported to our solution.

Library Detection In our current implementation, we assume that the integrated advertisement library is known beforehand. Given that the in-app library identification problem has been proved to be solvable with a high probability in LibScout [53], we can greatly improve the usability of `CompARTist` by adding such a library detection module.

Callee-side Rewriting The remote advertisement in our solution is incorporated into the host application by replacing the intra-application advertisement invocations with proxy calls from our support library. However, library invocations triggered by reflection methods or from native code are not supported in our current caller-side rewriting setting. A possible solution is to shift our approach to callee-side rewriting, for instance, hooking the library interfaces to redirect them to our proxies.

UI Testing Automation While precautions are taken to prevent some common pitfalls from occurring in our automated large-scale evaluation, there are still some weaknesses in the UI exploration tool we utilized. As described in Section 3.6.1.3, *monkey* in theory should be able to specify the target application and restrict the exploration to the user interfaces of that package. However, we observed several times that our tests were interrupted by unexpected events, such as *monkey*-generated touch events that overstepped the application UI boundaries during testing, disabling the device's

USB debugging option, or even going as far as factory-resetting the device. It seems that even though *monkey* is sufficient in presenting the feasibility of our approach, the existence of these undesired behaviors can introduce uncertainties to our testing. The current test infrastructure can be more reliable if a superior UI exerciser tool is provided. Other dynamic exploration tools like Brahmastra [54] and *DroidMate* [55] could be the possible alternatives for *monkey*.

3.6.5.2 Research Prospects

In addition to advertisement compartmentalization, `CompARTist` can also serve as the foundation for many future research projects.

Library Hotpatching A handful of top advertising networks, such as *Google Play Services Ads*, dominate the advertisement library market, so it is inevitable that many applications will statically integrate the same popular advertisement library packages, which leads to a considerable amount of library duplication between different applications. Generally speaking, library update work is a task for application developers. However, recent statistical work [53] has pointed out that developers often postpone, or in the worst-case scenario, simply ignore these updates. A potential solution to this problem is that we first provide an adapted instance of each library, each running in a dedicated application environment, and then transform the original local static library calls to references that dynamically link to the remote libraries. We've already shown that our system has the capability of accomplishing this transformation task, hence, we only need to create and maintain a system-centric repository of advertisement libraries. This system-centric repository further eases the distribution of library updates. When a new library update comes, all applications containing that library can be upgraded as soon as the library in the system-centric repository is upgraded, which is completely transparent to the app developer. The difficulty lies in that there may be backward compatibility issues with the library updates, but in any case, this system can at least be utilized to deliver security patches that have no public interface modification.

Beyond Advertisements As mentioned before, the loosely-coupled nature and well-defined interfaces of advertisement libraries make them more suitable for applying our approach than deeply-coupled libraries like Guava. However, there is also the possibility of applying our approach to isolate other types of third-party components in-between and even some new usage scenarios. For example, using our approach to compartment untrusted components or to deploy a system-centric update distribution solution.

3.7 Conclusion

This work introduces `CompARTist`, a compiler-based library compartmentalization solution to remedy the unsatisfactory situation of privacy and security threats induced by untrusted advertisement libraries. Our solution splits the original app into host and advertisement library components and moves the library to a dedicated app to create a strong security barrier. We apply inter-process communication and lifecycle synchronization to seamlessly reintegrate both components without impairing user experience. Our evaluation proves the robustness of our approach by successfully applying our transformation routines to 3257 apps from the Google Play Store. In conclusion, we introduce a new approach to library compartmentalization that abstains from system or app modifications.

4
Up2Crash

Evaluating Third-Party Library Updatability on Android

4.1 Motivation

As described in Chapter 3, the privacy and security threats originated from overly curious or even malicious third-party libraries can be greatly mitigated by establishing a clear trustworthy boundary between the library and the host and managing different entities with separate privileges. However, many libraries are not suitable for such a mechanism as shown in Chapter 3. In light of this fact, we instead in this chapter focus on the library itself, attempting to minimize the privacy and security risks associated with the vulnerability of libraries while retaining the original in-app library integration model. A number of studies [56, 57, 53, 58] have already looked at the problem of vulnerable libraries for Android applications and have shown that vulnerable third-party dependencies are actively in use within applications, e.g., a surprising ≈70% [59] of vulnerable free apps owe their vulnerabilities to integrated libraries. The most straightforward countermeasure against such vulnerabilities is to apply updates: a third-party library provider releases a fixed version, and applications that include the vulnerable version of the library can then be fixed by promptly updating the library. Unfortunately, due to the update bottleneck in the Android library update ecosystem, most library updates are not delivered to apps in this smooth manner as described in Section 2.3. Recent studies of third-party library updates [2, 60] confirm that most developers do not consider library updates as the reason for application version increment. Developers tend to preserve the outdated versions of the library to avoid the extra effort required for resolving incompatibility of their apps' code with newer library versions. Investigation of vulnerable apps' lifetimes [61, 62] also reveals the lack of incentives for non-functional updates. Given this situation, an automated update mechanism might be a way out of this dilemma.

4.2 Problem Description

Prior work [2] proposed the idea of an automatic update mechanism based solely on API compatibility between different library versions. Its statistics show that with such an automated library update setting, 85.6% of the libraries have at least one higher version available for update and 48.2% can even be updated to their latest version without any additional host code adoption. The problem is that the achieved updatability rate is derived solely from the results of static app analysis, which can only provide a glimpse of automated library updating from a theoretical and syntactic perspective. It ignores potential factors of version incompatibility that can immediately come into mind, such as obsolete APIs, intra-function changes, or secondary dependencies. So far, no ground truth exists about the existence and severity of those additional factors.

To bridge this gap, we try to answer in this paper the open questions *"What is the actual library updatability?"*, *"Do the updated libraries exhibit incompatibilities that prevent an easy drop-in replacement of library versions?"* and *"What are the primary causes for those incompatibilities?"*. To answer those questions, we opt in this work for studying apps' runtime behavior before and after applying drop-in replacements of API-compatible library updates. The best approach to do so could be **1)** an implementation of an automatic library updating solution and **2)** behavioral profiling of apps' runtime for both the original app and the one with library updates deployed.

Several existing works have dug into the problem of *patching* vulnerabilities in existing applications, such as Appsealer [63], PatchDroid [64], or Instaguard [65]. Unfortunately, none of them specifically focuses on library code. PatchMan [66] considered libraries, but only takes system libraries into account. Most importantly, the setting for a library updatability solution, which has to consider multiple update candidate versions, code changes beyond *simple* function-level changes, and potentially entangled dependencies (see Section 4.8.2), differs a lot from vulnerability patching solutions (e.g., a static rewriting solution cannot deal with entangled dependencies, or in-memory patching is limited to very local, small changes). Thus, none of the existing solutions is applicable as a suitable solution to the automated library updatability problem.

4.3 Contributions

To extend the status quo and investigate in-depth the proposed drop-in replacement of API-compatible library versions, this work presents a two-stage experiment. In this experiment, an automatic drop-in replacement library update framework based on classloader customization is put forward in the first stage, and then, in the second stage, dedicated, dynamic tests are carried out to evaluate the runtime behavioral differences between the original app and the one with an updated library. To the best of our knowledge, this work is the first to investigate the semantic problems and consequences for Android library updatability in a real-world setting in contrast to the previously estimated numbers purely on syntactic updatability.

Our study focuses on three popular, previously studied libraries (*OkHttp*, *Facebook SDK*, *Facebook Audience*). Our dynamic analysis results revealed that at runtime 4.08% (success rate 95.92%) of the tested updates experienced crashes after the drop-in library update. We discovered that multiple factors impede the automatic integration of a compatible library version. Through a source code study of crashed library versions, we discover incompatibilities beyond the public API, including deprecated public methods, changed data structures and library initializations that are only documented in the library changelogs, or entangled dependencies between the updated library and other libraries or the host app. Further analyzing the source code of 1,430 versions of 44 libraries showed that those discovered impeditive factors are prevalent in all kinds of other libraries and the claimed library updatability rate by prior works [2] should be adjusted. To provide a clear understanding of the library updatability, we re-calculate the updatability rate on a set of 332,432 apps after considering all those discovered factors. The comparison result shows that for *OkHttp* and *Facebook SDK* the picture is rather bleak, and their updatability rates sink 93.40%$\searrow$45.45% and 94.06%$\searrow$53.69% in the worst case, respectively, in comparison to previous estimates.

Thus, our work confirms the *technical* feasibility of an automatic drop-in replacement for library updates, but our test results also clearly show the existence of impeditive factors that prevent a drop-in library update from working correctly *in practice*. We think that our results provide valuable insights for the design of projected library update solutions that are independent of the app developer (e.g., drop-in replacements at the market or on-device) as well as for solutions that want to support app developers in maintaining up-to-date dependencies (e.g., through an IDE extension). We summarize our contributions as follows:

API-compatibility based library update framework To measure the realistic gap for drop-in library updates on Android, we first need a library update framework that follows the state-of-the-art proposal in prior work [2]. This work is first to present the design and implementation of a drop-in based Android library update framework. With this framework, a new library version can be opted into the original app at app launching time and be used as a replacement for the previous library version, which enables us to hunt library update-related runtime mal-functions further.

App runtime behavior profiling Using our library updating approach, two kinds of dynamic tests are carried out on real-world apps to not only validate the feasibility of our updating solution but also study the actual feasibility of drop-in library updates and re-evaluate the results of static app analysis in existing work [2]. By profiling the runtime behaviors of apps before and after library updates, we detect the occurrence of malfunctions introduced by the library update despite the library versions being API-compatible.

In-depth study of the obstacles for functional drop-in replacements By analyzing the malfunctioning cases, we discovered several factors brought by library evolution that prohibit the drop-in replacement of a target library to be functional. Based on those discovered factors, a follow-up study is conducted to evaluate the prevalence of those impeditive cases in other libraries. Our results show that those impeditive factors are important considerations for future solutions that target automatic library updates or that support app developers in their task of updating libraries.

4.4 Related Work

In this chapter, we introduce the related software patching techniques and the common test input generation tools for app behavior evaluation on Android.

4.4.1 Software Patching Techniques

Apart from going through the standard update chain described in Section 2.3, an Android software can also be fixed by third-party patches and application autonomous hotfixes.

Third-Party Patching Third-party patching reduces the vulnerability window of software as much as possible. Since the patches or patching framework are released by neither software developers nor official sources, they are not bound to the standard release procedure and can be deployed to fix software more efficiently. Patchdroid [64] applies in-memory patching techniques to update both userspace native code and Dalvik bytecode at runtime. Embroidery [67] uses both static and dynamic rewriting techniques to patch vulnerabilities in the Android framework and kernel. To be resilient against Android fragmentation and ensure system functionality across devices, Embroidery rewrites binaries at code-line granularity. With *reference hijacking* [66] the underlying system libraries are patched by redirecting library references to security-enhanced alternatives. InstaGuard [65] takes advantage of debugging features to enforce rules that block the vulnerability exploitation and avoid injecting new code while patching. KARMA [68] establishes a multi-level adaptive patching model to filter malicious input to the kernel. Appsealer [63] alters an app's intermediate representation to mitigate component hijacking attacks through a patched app version. None of the above solutions focuses on patching third-party libraries inside user apps. OSSPATCHER [69] targets at third-party libraries, but only open-sourced C/C++ libraries are concerned. There are also more works [70, 71] that automatically generate patches from source code. However, they do not apply to libraries included in applications that are usually not open sourced. Most recent work [72] rewrites app code to provide a library updating and sharing solution which is distinguished from our incompatibility root cause investigation purpose.

Application Autonomous Hotfix Application autonomous hotfix is a technique for self-healing apps in which the fix of the app is applied at runtime by the application itself. A number of hotfix frameworks [73, 74, 75] have been proposed to ease the distribution of *minor* patches. In those solutions, an official patch is first delivered to the app, and then the patch code is dynamically loaded into memory instead of outdated code. There is no need to reinstall the target app. With autonomous hotfixes, small fixes can be distributed to users swiftly without any user disturbance or central distribution point (e.g., Google Play Store). Unfortunately, those hotfix plugins are required to be integrated by app developers and the patches should be released by them as well, which highly depends on developer incentives and is not applicable to efficiently update libraries within already existing apps. However, the flexibility of those dynamic code integration techniques and plugin techniques [76] is quite inspiring and our third-party library updating solution is established based on them.

Patching vs. Updating Most of the patching solutions use techniques, such as static rewriting, in-memory function patching, or vulnerable path blocking, to mitigate vulnerabilities. However, the scenario for library updatability includes but is not limited to rolling out those pinpointed code fixes that are prevalent in patching scenarios. Library updates usually concern not only intra-function changes, but also inter-function changes, secondary dependency updates, and resource file changes, especially when upgrading across multiple versions. For this reason, a full library drop-in replacement update exceeds highly localized patching as described in the existing works. Prior work [66] also applies full library replacement for system libraries. However, the statically integrated third-party libraries in apps, in contrast, vary from app to app and in their versions, which prohibits a central, system-wide replacement of a third-party library. Furthermore, our paper studies the problem of library updatability and not specifically of *patching security vulnerabilities* since for the mobile library ecosystem, prior work [2] reports that security and privacy patches are unfortunately commonly mingled with minor/major releases, and unfortunately very few library developers report security and privacy relevant changes in their logs. There is an expected high dark figure of *silent* patches. Thus, patching security and privacy issues of libraries currently boils down to keeping library dependencies up-to-date. Our work tries to investigate the root causes of incompatibilities in this process for auto-updates.

4.4.2 Android Test Input Generation Techniques

We utilize Android test input generation techniques to capture potential differences in the app behavior after applying library updates. The existing exploration engine for testing tools can be categorized into three types: random, model-based, and systematic.

Random Exploration Testing tools with a random exploration engine explore an application with generated semi-random sequences of events. Both Android's official UI exerciser *monkey* [47] and the open-sourced *DroidMate* [3] are equipping with a random exploration engine. We here apply *monkey* to our large-scale experiment and apply *DroidMate* to our runtime behavior profiling test. Despite the limitations that complex tasks like logging in to an account are unlikely to be performed by random exploration tools, existing work [77] has proved the effectiveness of these approaches. In this work, we extend *DroidMate* with a plug-in that hard-codes some specific exploration actions, such as registration and login to applications, so as to improve the overall exploration efficiency (see Section 4.6.2.3).

Model-based Exploration Model-based exploration tools take advantage of the results of static and/or dynamic analysis of the target application to build models and create test cases. Among these tools, GUIRipper [78] explores an app's GUI structure and generates the app's behavioral model in a state machine representation. SmartDroid [79] utilizes static analysis to extract paths for further dynamic exploration. *DroidMate* [3] is primarily dependent on dynamic analysis during which it obtains the app model. By re-identifying UI elements with this model, *DroidMate* can prune out the re-exploration of known UI elements and continuously guide the test to new ones.

Systematic Exploration Systematic exploration tools combine various algorithms to traverse the application as far as possible or to generate test cases based on certain expectations. Sapienz [80] employs search-based algorithms and random fuzzing to achieve better test coverage. IntelliDroid [81] applies symbolic execution in test event generation to trigger desired behaviors. While these approaches excel in specific exploration scenarios, their reliance on static information makes them underperform in scenarios where the tested application involves external sources (e.g., web content), native code, obfuscation, or dynamic code (e.g., reflection). Another closely related work in this category is Brahmastra [54], which performs static analysis to gather paths to target third-party code and jump-start that code directly through app binary rewriting. While this approach significantly increases the probability of hitting target

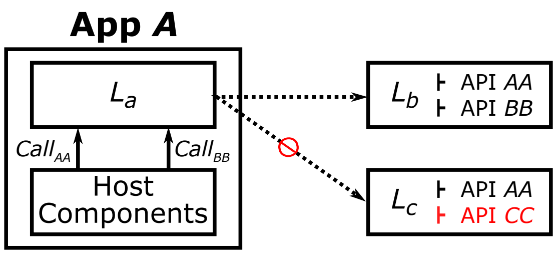


Figure 4.1: A typical scenario for API-compatibility based updatability

third-party code, it alters the application, thus may introduce additional uncertainty to the exploration results.

4.5 Requirements Analysis

Considering the alarming rate of outdated libraries and the inefficient third-party library updating chain explained in Section 2.3, we put our focus in this work on evaluating the runtime library updatability situation under an automatic third-party library updating framework, as well as tracing the root causes for potential side-effects brought by updating.

Typical scenario for API-compatibility based updatability

Existing studies have highlighted thrilling API compatibility across different library versions. Here, we describe a typical scenario based on Derr's et al. work [2] and their *LibScout* tool (see Figure 4.1). App A contains library L in version a with invocations AA and BB. If interfaces AA and BB still exist in the successor version L_b, but only partially exist in version L_c (e.g., parameters or types of a method have changed or a method was removed), *LibScout* reports library L_b as compatible with library L_a inside App A but not L_c.

Implementation of an automated library update framework

To investigate the updatability and catch potential incompatibilities beyond the theoretical results of prior works, we need a library update framework that follows the methodology proposed in the existing studies. In our paper, we follow the proposal of Derr's et al. study [2]. Given the scenario above, an implementation of an automated library update framework should try to update library L from version a to version b. Another precondition of this API-compatibility based library updating solution is that the update should be a drop-in replacement and no host code adoption performed. The library upgrade could be done before or after app build without new host code adoption. In our work, we focus on a post-build upgrade, because compared with a pre-build upgrade, which is done through IDE plugins by app developers, a post-build upgrade is more flexible and can deliver the updated library version promptly, circumventing the upgrading bottleneck brought by the developer-dependence. Considering the complexity of the library updating scenario, which can include changes, such as inter-function code changes, secondary dependencies, or resource files, a naive static rewriting solution would cause an immediate crash/misbehavior (e.g., app failed to log into Facebook when the app's signature was changed by static rewriting), which is then detrimental to exploring update incompatibilities. To try our best to eliminate unnecessary interferences and explore incompatibilities as reliably as we can while upholding conditions proposed in prior work [2], here we borrow the idea of opting in codes by classloader customization from existing frameworks [73], [76] and carefully design a dynamic library drop-in replacement framework (with secondary dependencies included) to support automatic library upgrading across both minor and major versions.

Automated library update testing The API-compatibility based library updatability results presented in previous work [2] are based on static app analysis, which can only reflect a theoretical and syntactic situation. To understand the actual feasibility of an API-compatibility based library update, further runtime testing is necessary. The most established dynamic app testing is automated user interface (UI) testing, which performs a series of UI operations on the target app. By doing so, the app behavior can be profiled, and potential failures and dysfunctional behavior after library updating be discovered by comparing the runtime profiles of the original and the updated app. It is noteworthy that the feasibility of our library update framework can be confirmed in this context since behavioral correctness is a strict baseline for our testing.

4.6 Two-stage Updating Experiment

The goal of our study is to evaluate if a simple drop-in replacement update is a viable option to solve the problem of outdated libraries on Android. In this section, we describe a two-stage experiment to test apps' runtime behaviors before and after a library update. In the first stage, we apply an automated library updating framework that we developed according to the proposal of prior work [2]. This framework allows replacing an outdated library inside an app with a newer version without additional host code adoption. During the second stage, two automated user interface (UI) tests are performed to evaluate the behavioral correctness of target apps after drop-in replacements of library updates. This approach allows us to report on the gap between the theoretical updatability rate in the literature and the actual runtime rate and its impeditive factors.

4.6.1 Stage-1: Automated Library Update Framework

To support automated library updating without host code adoption, this work implements a dynamic updating framework that takes advantage of the class domain isolation and dynamic code loading features of Android's classloader hierarchy. The outdated libraries are automatically updated at app load-time by loading the new library from a well-defined place by a customized classloader, and in this process no additional code adoption is required for the host app's code.

The framework is composed of three modules as shown in Figure 4.2: `Update Execution Environment`, `Update Handler`, and `LibCenter`. `Update Execution Environment` is established on a customized build of Android, which is extended

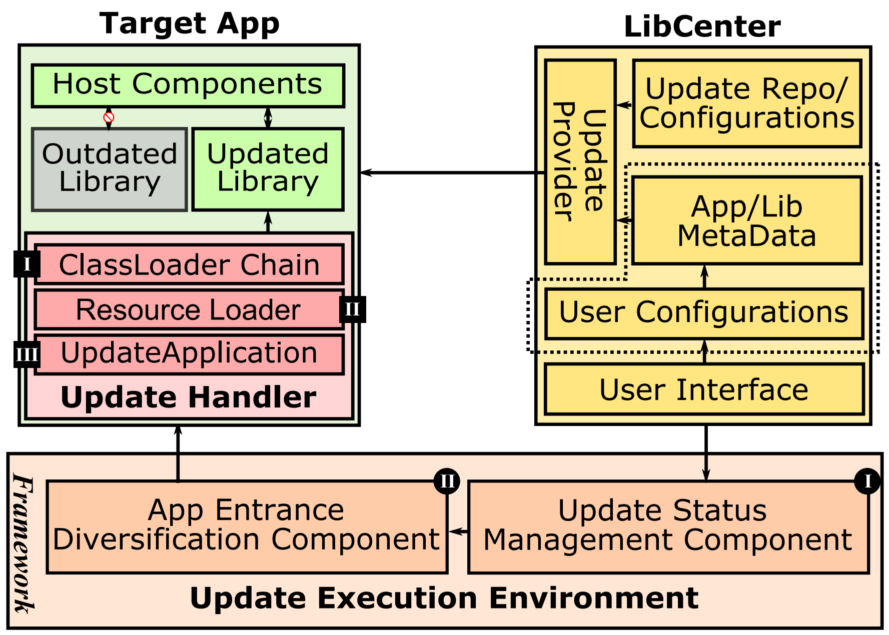


Figure 4.2: Overview of Library Update Framework with three modules: Update Execution Environment, Update Handler, and LibCenter

with components to support library updates. `Update Handler` is a customized classloader chain together with auxiliary components for applying library updates at app load-time. This customized classloader chain isolates the loading of library code and host components at runtime. As a result, the library update can be opted-in as a replacement of the original library by solely altering the library class loading path. `LibCenter` is the centralized library management module. All the library updates and included library information for installed apps are maintained by it. It is also the user interface for update configuration. Through this app, the library update for a target app can be configured and delivered to the target app. Together those three modules enable automatic distribution and application of library updates without developer support and ease our testing by allowing us to flexibly roll-out library updates to the installed apps-under-test.

4.6.1.1 Update Execution Environment

To update a library of an app, the updated version should be available to the app. However, we have to abstain from modifying the app to avoid malfunctions due to induced bugs and also to adhere to the proposed methodology we are testing. Thus, the system should opt in the updated version before app initialization, which we accomplish through an `Update Execution Environment` as an extension to vanilla Android. This environment consists of two key components (see also Figure 4.2): I an `Update Status Management Component` to maintain a global update status of apps; and II an `App Entrance Diversification Component` to enable library updating for an app. The internals of `Update Execution Environment` are illustrated in

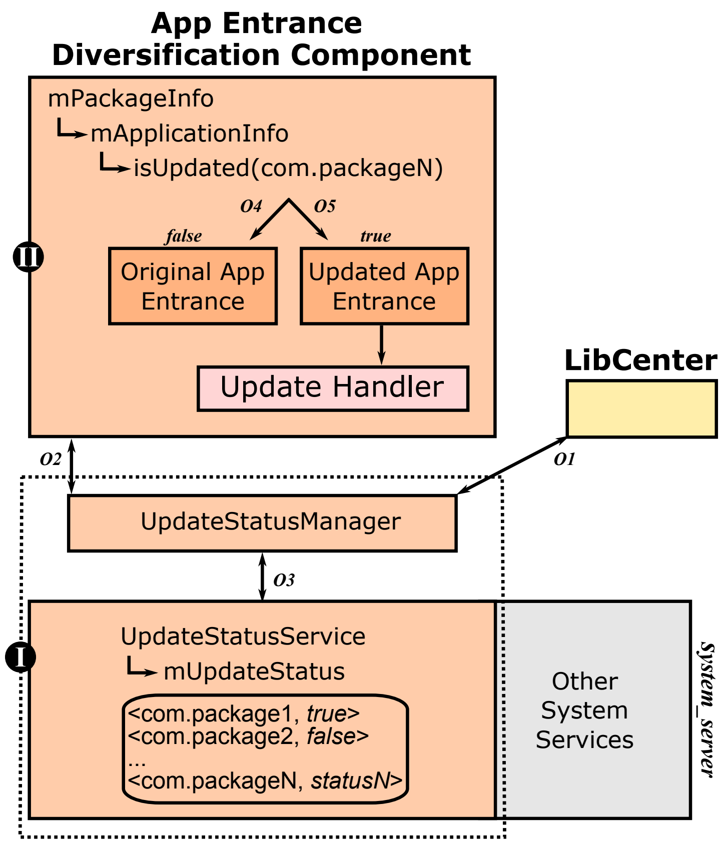


Figure 4.3: Update Execution Environment

Figure 4.3.

I `Update Status Management Component` manages the update status for each app and allows us to control if an app runs with its original or an updated library version. Its `UpdateStatusService` is a dedicated system service that records each app's update status according to update events sent by `LibCenter` and unifies the update operations from system-side in the II `App Entrance Diversification Component` and the update configuration from user-side in `LibCenter`. Client processes can reach the service over *Binder* IPC via a custom manager, `UpdateStatusManager` (***O3***), to set and get the update status for each app. `LibCenter` sets the status of target apps (***O1***) and II `App Entrance Diversification Component` retrieves (***O2***) at app load-time the status for the loading app to determine which library update actions should be taken.

II `App Entrance Diversification Component` is the actual update deployment site and takes care of loading the updated library version into the application process. As can be seen in Figure 4.3, `App Entrance Diversification Component` is implemented as a customized app launching process with an additional `Application` class interface. The `Application` class is the first class loaded in each app's process life-cycle. Initialization of the app and of the included library is usually executed inside the `Application` class to ensure they take effect at an early stage in the process' lifetime. Through this `Application` class, we added a new app entrance to an `UpdateApplication` (Section 4.6.1.2) to the original app launching process so as to control which library version should be loaded during app launching based on the updating status gained from the ❶ `UpdateStatusService`, which was set via `LibCenter`. With this modified launch process, the target app can switch between the original library version (***O4***) or the updated version (***O5***). In case of a library update, `Update Handler` continues the app launch process.

4.6.1.2 Update Handler

As shown in Figure 4.4, `Update Handler` is a bridge connecting host app and library update and is responsible for activating a library update for the app. To ensure any app on our modified Android can activate library updates, we integrated `Update Handler` into the Android framework as a static library that is automatically loaded into all app processes. `Update Handler` is composed of an I `Update ClassLoader Chain` for separating the target library code loading from the rest of the app code loading, an II `Update Resources Loader` to attach resources of the updated library to the original app, and an III `UpdateApplication` to activate `Update ClassLoader Chain` and `Update Resources Loader` before the initialization of the app-under-test.

I `Update ClassLoader Chain` is a customized classloader chain specifically for dynamic library updating. Android inherited Java's parent-delegation mode in which a series of classloaders are chained together and each non-root classloader will delegate a class loading request to its parent classloader first before loading the requested class by itself. Only the root classloader will try to load the target class by itself directly. This parent-delegation mode separates the loaded code into different security domains according to their path, which prohibits a low priority classloader from exposing high priority code. For instance, `PathClassLoader` is in charge of loading installed application classes (class path in */data/app/package.name*) and cannot load non-installed packages (e.g., class path in */sdcard/*). Same class loaded by different classloaders is treated as different classes and cannot be cast to each other. In our design, classes from the updated library version should be loaded instead of the original outdated ones. However, the app package, including both the app code and libraries code, is a fixed bundle and the classes inside a user application are in general loaded by Android's default `PathClassLoader`. To suppress the loading of the originally contained library and opt-in the classes of the updated library, a new classloader chain is introduced in our design to isolate the loading of the updated library from the host application. Different with existing classloader customization based patching solutions [73, 82] which replace all outdated classes to updated ones directly, our solution constructs an isolated container for the interaction between updated library and its updated dependencies. Thus, both of the original and updated secondary dependencies are preserved in this design while updating the target library (first dependency) so as to provide better updating compatibility for cases where host codes involve invocations to secondary dependencies. Figure 4.4 shows how the two classloaders are customized for this new classloader chain.

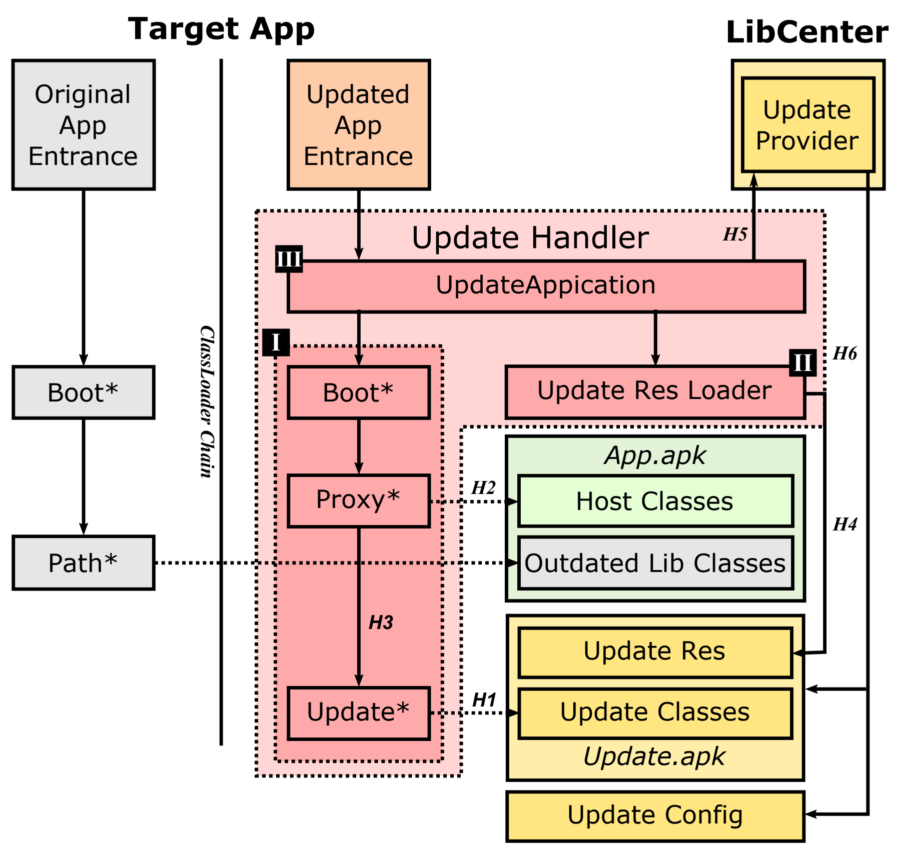


Figure 4.4: Update Handler (Suffix * indicates ClassLoader)

UpdateClassLoader is an extension of BaseDexClassLoader, which is capable of loading *dex* files from a designated path. It is responsible for loading updated libraries without additional app code merging (***H1*** in Figure 4.4). This update-specific classloader is independent of the update, i.e., as soon as a newer library version becomes available, that version can be integrated with the host app by simply replacing the library file for updates and without touching the app package itself. However, objects created by different classloaders are not available to each other, which could complicate the interaction between the library and host application. To alleviate this problem, UpdateClassLoader has to be a node in the system classloader chain. It is linked as a child to ProxyClassLoader, the newly created classloader for application code.

ProxyClassLoader is an extension of PathClassLoader, which can only load installed applications files. Apart from loading the updated library's classes, the original host application should also be loaded. The app code is simply loaded by the default PathClassLoader. However, the original library code is intertwined with the host components inside the original app package (i.e., *dex* file). The original library code will also be loaded automatically by PathClassLoader when being invoked by the host components. To create a clear boundary between the host components and the library code that should be replaced with the updated version, ProxyClassLoader is constructed to delegate the loading of all updated library classes to `UpdateClassLoader`. To minimize the impact of this modification, `ProxyClassLoader` is initialized on the basis of the original `PathClassLoader`. Everything of `PathClassLoader` is preserved ***(H2)*** except for an additional class name filter when loading classes. When the class to load is from the target library, the name filter in `ProxyClassLoader` will distinguish the library package prefix in the class name and the loading request for this class is delegated to `UpdateClassLoader` ***(H3)***, which will finish the class loading ***(H1)***. This way, the original, tightly integrated library will be replaced at app loading time with the newer library version.

II `Update Resources Loader` integrates the resources of the updated library ***(H4)*** into the app. Though not all libraries require additional resources, still a large fraction does in order to enhance their functionality, e.g., *Facebook SDK* requires resources to customize the login button. Since Android resources are labeled with a 32-bit ID, there could be id conflicts between the original app resources and resources of the drop-in library. Our solution is to compile the library update within a wrapper application (*com.wrapper*) as a shared library, so the generated resource ids will not be constants and can be reassigned to a separate range at runtime. To enable the usage of resources inside the added library update, its resources should be attached to the app space through `addAssetPathAsSharedLibrary` interface. Since the assigned IDs for the new resources might differ from the resource ids used with the library code, we rewrite all of the individual library *R* classes with values in the merged resource file from wrapper package. After that, the new resources are available to both the library code and host application and no id collision can happen between the original and the new library resources.

III `UpdateApplication` is a customized `Application` class. The main idea is to ensure the updated library is activated before any host application code takes control. Considering that some library initialization is by default done in app's `Application` class, the activation of the new library should be handled before that. The most convenient and least intrusive solution is to hook the application initialization process by replacing the original app `Application` class with `UpdateApplication` class that is described in Section 4.6.1.1. After the replacement, the system will treat it just as the original `Application` class and finish the application initialization process. In this initialization process, a request for library updates will be sent to `LibCenter` from the target app's process space ***(H5)***. `LibCenter` will return an authorized URI that can be used to copy the library package and configuration files to the target app's storage ***(H6)***. Furthermore, the creation and initialization of both I `Update ClassLoader Chain` and II `Update Resources Loader` are also accomplished here based on the files retrieved from `LibCenter`. Last but not least, the newly generated classloader chain is enabled and the application can be launched as usual. To minimize system modifications in integrating the new classloader chain, we follow the classloader hooking approach used in former works [73, 76].

4.6.1.3 LibCenter

`LibCenter` acts as a centralized library repository from which library updates are retrieved. All pre-compiled library packages and metadata are stored here. Using

LibScout[1] as part of `LibCenter`, we collect all installed apps' metadata including information about used libraries and library compatibility information based on the library API calls from the host apps. `LibCenter` uses that information together with user preferences set via `LibCenter`'s UI to create linking information about which API-compatible library update can be exposed to the `Update Handler` in target apps through an `Update Provider` (see Figure 4.2).

Precompiled updates in `Update Repo` are a set of wrapper applications containing different versions of different libraries. Once an update is activated, its corresponding wrapper application will be copied to the app's process space so the updated library version inside the wrapper is available to the target app (see description earlier). The generation of wrappers for each library version is automated using *Gradle* [83] build tool with a template app. By altering the dependency library information in the *build.gradle* file of the template app, *Gradle* can synchronize the specific library version from its central repository and build the final wrapper application for this library version. There are two advantages in wrapping the updated library in an application with *Gradle*. First, considering that those target libraries also need their own dependencies, e.g., *OkHttp* depends on *okio*, we automatically bundle the target library together with its dependencies to avoid conflicts between the newly added library and the original library dependencies. Second, by wrapping the library bundle into an apk file, the resource file *R.java* can be generated and automatically arranged with *aapt*[2], which is necessary when invoking library calls that need resources. Here we compile libraries as shared libraries to avoid resource id conflicts as described earlier.

`Update Configurations` are a set of files that describe the generated wrapper packages. As mentioned in Section 4.6.1.2, information such as library class prefix and resource classes are necessary for correctly loading library code and resources. `Update Configurations` carry all the requested information of a library update and are sent to the target app together with the library package.

`User Interface` allows personalized settings for library updates, e.g., the target app, target library, and update version and is used by us to set up our test scenarios.

Update linking rules are created to dispatch a proper library update to a target app. They depend on both the *LibScout* generated library `API Compatibility Metadata` and `User Configurations` (e.g., targeted library version). Library `API Compatibility Metadata` records the relationship between the host application and target libraries gained from offline library detection. For example, in the scenario described in Figure 4.1, a profile for the relation between app A and library L version a will be created in a form of quintuple *[A, A's version, L, a, [b]]*, where *[b]* is the list of API-compatible library versions. `User Configurations` designates the target app and library as well as the target library version. Combining the quintuple and user preference for an app, `LibCenter` can link a specific library update to the target app. This linking information will be used by `Update Provider` for exposing the correct wrapper application to the target app.

`Update Provider` is simply a `FileProvider` to share files, here wrapper applications, between target apps and `LibCenter`. It uses `Intents` containing the URI for

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

[2]https://android.googlesource.com/platform/frameworks/base/+/master/tools/aapt

the corresponding wrapper application in response to requests by `Update Handler` to allow `Update Handler` to retrieve the library update from `LibCenter`.

4.6.2 Stage-2: Automated User Interface Tests

In the second stage of this experiment, we choose top ranking libraries as our case studies for library updates and run multiple dynamic tests on real-world apps from Google Play Store that include those libraries in order to have a close look at apps' runtime behaviors after API-compatible library updates. To ensure the comprehensiveness of this experiment, firstly, we carried out a large-scale dynamic test to provide a macro-view of not only the feasibility of our update framework but also of immediate malfunctions, like crashes, in target apps brought by those drop-in library replacements. Second, we execute a more intensive test to explore more app functionality so as to trigger more hidden malfunctions introduced by the updates, e.g., changed side-effects of library methods, although a full anomaly detection is beyond the scope of this paper.

4.6.2.1 Target Libraries & Apps

Different libraries have different integration approaches with their host apps. We carefully select three libraries, with 78 library versions in total, from different library categories [84] as target libraries: *OkHttp* from Development Tools, *Facebook SDK* from Social SDKs and *Facebook Audience* from Ad Networks. Those three libraries are the most popular libraries from reputable companies which are well-maintained and include secondary dependencies. Instead of targeting more libraries, the experiment setting here is more to utilize limited dynamic testing in highlighting a lower bound on the existence of incompatibilities when considering various library versions. To compile a list of apps that contain those libraries, we run *LibScout* on an app repository containing 332,432 free apps crawled from Google Play Store with 128 library profiles for three libraries from the *LibScout* project. We found 379,429 library-app pairs. *LibScout* can not only provide a list of apps that contain a target library but also the detailed API usage of the library. To make the evaluation more comprehensive, we extend *LibScout* with a ranking module to cluster apps into different sets based on the library APIs invoked in the host app components. For dynamic testing, we select 3,000 apps, 10 apps from each of the top 100 frequently used API sets for each target library. There are 78 library versions (25 from *OkHttp*, 33 from *Facebook SDK*, 20 from *Facebook Audience*) in our final data set.

4.6.2.2 Monkey Test

Our update execution environment is deployed on Android version 7.0. We test our framework on two Pixel C devices that are flashed with our customized system. In this large-scale evaluation, we try to update each library to the latest, API-compatible version. To measure the hit rate of (updated) libraries during testing, we used *soot*[3] to inject log statements into the frequently invoked interfaces of each library. To better

[3]https://github.com/Sable/soot/wiki/Tutorials

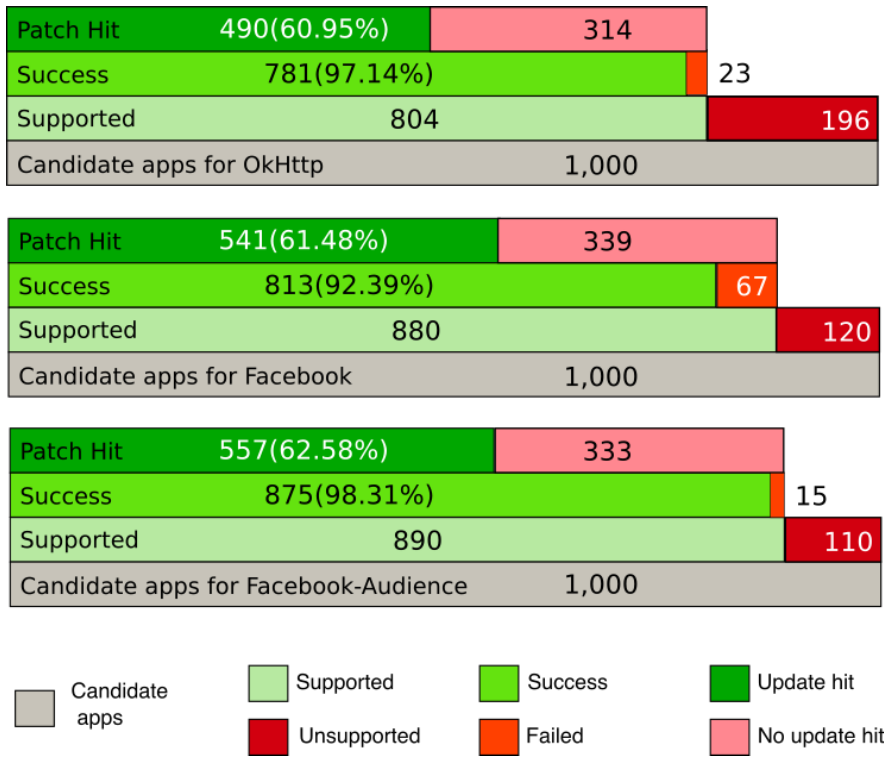


Figure 4.5: Monkey evaluation results for apps including OkHttp, Facebook SDK , Facebook Audience

scale the dynamic testing, we chose the open source *monkey-troop* tool[4] to install test apps and execute them using Google's official application exerciser, *monkey* [47], on both devices in parallel and automatically. In each monkey run, 500 random events will be explored. During this process, the execution log is recorded for measuring the library hit rate.

Ground truth Considering that some apps could be malfunctioning for reasons like failed download, obsolete APIs for our test platform, buggy design, etc., we first run *monkey-troop* on the original apps. Only if the first run executes successfully without errors, those apps remain in the test app set and are considered for library updates.

Test results UI exploring results for the 3 libraries are shown in Figure 4.5. The uppermost graph describes the update result for *OkHttp* library, where 781 of 804 supported tested apps passed *monkey* without a crash, giving a success rate 97.14%. Among all those apps supported, 60.95% of them hit the updated library. For *Facebook SDK* library, 813 of 880 supported target apps did not crash during monkey exploration, resulting in a 92.39% success rate, and 61.48% of those supported cases hit the library. The result for *Facebook Audience* is quite similar with a 98.31% success rate for the 890

[4]https://github.com/Project-ARTist/monkey-troop

supported apps and a 62.58% hit rate. From all those results, we can see a success rate of 95.92% in total, which is close to ideal. However, the overall hit rate of 61.69% is not very inspiring. More than 38 percent of apps passed *monkey* without hitting the library. The reason could be, for instance, that the library is integrated at a hidden position, which cannot be hit easily (e.g., the target library is only invoked after purchasing a product successfully), or the *monkey* failed to explore the specific path (e.g., clicking at a specific position on screen to jump to another page). Those are the common limitations of large-scale dynamic testing with random exploration by *monkey* and have already been noticed in various existing works. To complement our results, a more in-depth but also time-intensive testing based on *DroidMate* is conducted to evaluate the internal misbehaviors of the host app after library updating.

4.6.2.3 DroidMate Test

In our second test, we randomly select 5 applications from the *monkey* test set for each library and run *DroidMate* on all 15 apps to trace their runtime behaviors before and after applying library updates. By comparing the *DroidMate* tracing results for the original application and the app with library updates, we can catch more intricate behavioral changes other than crashes. More specifically, we treat the blocks of an app's source code containing libraries as app functionality and a variation of the blocks reached before and after applying library update as behavioral change. Whenever the library update alters the app's original execution routines (e.g., exception handling routines), the tracing results for the updated app will deviate from the original one, and the behavioral change caused by the update can be noticed and reported by *DroidMate*. As mentioned in Section 4.4.2, *DroidMate* has limited support for complex tasks, such as app registration and login. To further improve the exploratory capabilities of *DroidMate* and reach deeper application functionalities, we extend the *DroidMate* [3] platform[5] with a dedicated plug-in that would bypass the restrictions of the original *DroidMate* and *monkey* tool, for example, by providing an app's registration and login information. This second test is carried out on four Pixel C devices and four Pixel C emulators. Both the real devices and emulators are running the same customized version of Android 7.0 as the monkey-based testing. In consideration of test consistency and accuracy, the two runs of each app should be performed on the same device or emulator. Here we use the list of all possible blocks, which is gathered by instrumenting each test app with the open-source *ARTist*[6] [1], as the ground truth. Given the fact that some blocks may be unreachable by cause of our test configuration or app usage, this list may represent an over-approximation of the actual possible behavior. At test time, all executed blocks will be recorded, except for those from the target library. Block information for the target library can differ between versions by design, so we omit any reached blocks from the target library to keep test accuracy. Furthermore, we set a *library reached* tag for the target library. By monitoring the existence of this tag, we can check if the target library has been hit. Any runs that did not reach the target library should be discarded.

[5]https://github.com/uds-se/droidmate
[6]https://github.com/Project-ARTist

Table 4.1: Results of in-depth analysis with DroidMate plug-in (comparison between code covered for original and updated app)

App Name	App Version	Library	Original			Updated		Change
			App	Overall	St.Dev.	App	Overall	
Shalom Shalom Radio	2.0	OkHttp	18.59%	17.81%	3.17%	19.36%	16.76%	No
Maurin Hyundai	3.0.4	OkHttp	13.33%	48.93%	1.06%	13.33%	48.93%	No
Blur Effect Keyboard	1.185.1.102	OkHttp	31.36%	37.36%	2.87%	33.58%	35.85%	No
UK Online FM	1.0	OkHttp	48.35%	60.86%	0.71%	48.35%	60.46%	No
Sanimedius Apotheke	2.1.10	OkHttp	56.58%	37.94%	3.07%	57.89%	37.90%	No
LOOM CLUB	4.785	Facebook SDK	23.62%	29.03%	2.26%	20.25%	23.98%	No
Farmacia Charo Ferrá	0.01	Facebook SDK	11.76%	36.80%	4.16%	11.76%	34.62%	No
SnapOdo	0.1.0	Facebook SDK	11.76%	51.61%	1.24%	11.76%	46.53%	Yes
Close Up	2.2 Forest	Facebook SDK	58.60%	54.38%	0.17%	57.67%	54.26%	No
Stevenson Student Activities	5.63	Facebook SDK	42.72%	48.29%	0.46%	42.55%	46.91%	No
Metal Tombstone	4.1 Pea Green	Facebook Audience	60.50%	19.88%	0.12%	61.00%	19.65%	No
Personal Tracker	1.5	Facebook Audience	54.72%	30.54%	1.14%	64.15%	30.78%	No
Paris Metro Map	1.1	Facebook Audience	23.08%	25.72%	2.81%	23.08%	25.83%	No
Burak Yeter Songs	1.4	Facebook Audience	50.00%	50.00%	0.00%	50.00%	50.00%	No
Maquillaje Halloween 2017	13.0.0	Facebook Audience	24.49%	12.66%	1.55%	24.23%	12.56%	No

Test design For each app, we repeated the execution separately for the original app and the application with the library update until we had collected 10 runs each with library hit. Given that the exploration is random and the target app may contain non-deterministic content, such as advertisements, we choose to conduct 10 runs for each app version to mitigate the variability of testing while still maintaining a reasonable time trade-off for the overall test procedure. 500 events, including clicks, long clicks, swipes, etc., are applied on visible, enabled, and clickable UI elements at each run. During testing, our *DroidMate* plug-in first performs predefined actions such as entering user information and clicking the login/registration button to unlock further app screens. It then explores the target app with *DroidMate*'s standard biased-random approach, which assigns a higher priority to unvisited UI elements to increase the chances of discovering new paths and thus improve code coverage. To simplify the display of the results, we group the execution results into two categories: original (O) and updated (U). For each category, we put the blocks that have been reached by at least one run into his hit set (B_O and B_U). By comparing the intersection of these two sets ($B_I = B_O \cap B_U$) against the set of blocks only reached in the original runs ($|B_O - B_I|$), we achieve the behavior change between the two app versions. If this difference is greater than $3 \times$ *standard deviation* of the average coverage among the elements in B_O ($|B_O - B_I| > 3 \times \sigma(O)$)—which covers 99.7% of the values assuming the coverage variation follows a normal distribution—we considered that there was a behavioral change.

Test results Table 4.1 lists the test results. For each app, we collect application code coverage data, which includes only blocks within the same package as the application, as well as overall block coverage data that includes both application blocks and library blocks. These coverage data can indicate the depth and relevance of the test. According to Table 4.1, the *DroidMate* plug-in achieves an app block coverage of 35% on both the original app and the one with library updates, with a minimum of 11.76% and a maximum of 60.50%, and overall block coverage of 37% for the original apps and 36% for the updated ones, with a minimum of 12.66% and a maximum of 60.86%. Results show that both sets of coverage data fall within the range of expected coverage for state-of-the-art test input generation tools [77]. We use the *3 standard deviation* tolerance as the analysis metric and find out that only the *SnapOdo* app, which includes the *Facebook SDK*, is noticed a behavioral change. The further manual inspection of the app confirms that this app stuck at the Facebook login page after applying library update, thus reducing the number of reachable blocks at exploration. In addition, *LOOM CLUB* app also displays a significant coverage difference (5%), but the same discrepancy could be found in its original application run as well. Given its high degree of non-determinism, we regard this application as exploration noise.

4.7 Root Cause Analysis

Our two-stage experiment in Section 4.6 demonstrates the occurrence of app runtime behavioral deviations after API-compatible library updates and shows that library updating is not as straightforward as the existing work [2] claimed it to be. In this section, we deep-dive into the failure cases in our tests to study the factors that impede

Table 4.2: Categorized exceptions reported by Monkey test

Exception / **Error Message Example**	**#App**	**%Failure**	**Library**	**Version Original – Updated (#)**
AbstractMethodError	17	73.91%	OkHttp	3.0.0-rc1 – 3.9.0 (17)
AbstractMethodError: abstract method "void okhttp3.Callback.onResponse (okhttp3.Call, okhttp3.Response)"				
ClassNotFoundException	4	17.39%	OkHttp	3.2.0 – 3.9.0 (2) 3.3.0 – 3.9.0 (1) 3.3.1 – 3.9.0 (1)
NoClassDefFoundError: Failed resolution of: Lokhttp3/internal/Platform				
FacebookException	52	77.61%	Facebook SDK	4.0.1 – 4.26.0 (1) 4.6.0 – 4.26.0 (8) 4.16.0 – 4.26.0 (1) 4.1.0 – 4.26.0 (2) 4.7.0 – 4.26.0 (1) 4.17.0 – 4.26.0 (12) 4.2.0 – 4.26.0 (1) 4.8.0 – 4.26.0 (1) 4.3.0 – 4.26.0 (1) 4.8.2 – 4.26.0 (8) 4.5.0 – 4.26.0 (1) 4.9.0 – 4.26.0 (15)
RuntimeException: A valid Facebook app id must be set in the AndroidManifest.xml or set by calling FacebookSdk.setApplicationId before initializing the sdk				

This table only lists library related exception cases.
#App: the number of failed apps that reported this exception
%Failure: the percentage of apps that failed for this exception among all the monkey failures of this library

library updating.

4.7.1 Findings from Monkey Testing

We analyze the *monkey* logs of the failed apps and categorize all failures according to the reported exception messages. Though we did a pre-run on *monkey* for each app to filter out those apps with innate faults, a flawed app can still survive the first run and crash in the second run because of the random behavior triggered by *monkey*. Since we are not working on app debugging, investigating the failure reasons for all failure cases would be a wild-goose chase. Here, we concentrate only on the failures that have an obvious relationship with updating libraries. We consider all the failures that contain library specific keywords in their exception messages. Table 4.2 provides an overview of those failure instances. It can be observed that both *OkHttp* and *Facebook SDK* have interesting exceptions at runtime after updating them, while we discovered nothing of interest for *Facebook Audience*. For *OkHttp*, 17 apps failed because of `AbstractMethodError` and 4 apps failed because of `ClassNotFoundException`, which together make 91.30% of all failure cases for *OkHttp*. Library *Facebook SDK* has 52 apps throwing `FacebookException`, which equals 77.61% of all failures for that library.

4.7.1.1 AbstractMethodError

This exception is thrown when an abstract method is called but the definition of a target class, here class `okhttp3.Callback`, is incompatible with the currently executing method. All of the 17 crashes happened when updating *OkHttp* from version 3.0.0-rc1 to 3.9.0. Version 3.0.0-rc1 is the first version with the 3.x API. This is a breaking upgrade that even changed their package name from *com.squareup.okhttp* to *okhttp3*. Version 3.9.0 is the latest *OkHttp* version in our library repository. With all those background information and our test setting that libraries are always updated to their newest version among all the compatible versions, this is a strong indication for incompatible changes between those two library versions.

Source code analysis Library *OkHttp* is open source, and we investigate the source of `okhttp3.Callback` and find its evolution trace, which is shown in Listing 4.1. We find that in version 3.0.0, *OkHttp* modified the interfaces defined in `Callback` by taking an additional `Call` object as a parameter for both `onFailure` and `onResponse` interfaces to facilitate invocations to the `Call` object inside the `Callback` as described in its changelog. This change remained up to the newest version. This kind of mismatch should be detected as an incompatibility between versions, and its update should be disallowed in our test settings. However, *LibScout* detects library invocations via root package matching. Since interface implementations are usually named under a host package prefix (e.g., *com.host.package.Callback*), they are attributed as a host call by *LibScout* when invoking `onResponse` interfaces of a `Callback` host implementation and escape from the library compatibility check. To eliminate this kind of false positive cases, *LibScout* should also take the library's public interfaces into consideration. In

```java
// version 3.0.0-rc1  release date: 2016-01-02
public interface Callback {
void onFailure(Request request, IOException e);
void onResponse(Response response) throws IOException;
}

// version 3.0.0  release date: 2016-01-13
public interface Callback {
void onFailure(Call call, IOException e);
void onResponse(Call call, Response response) throws IOException;
}

// version 3.9.0  release date: 2017-09-03
// the same as 3.0.0
```

Listing 4.1: Evolution trace of okhttp3.Callback class

this case, all of the 17 apps will be non-updatable under these new constraints. Also, the claimed update rate by earlier work should be updated.

4.7.1.2 ClassNotFoundException

This exception is thrown when a classloader failed to load the target class by name in the classloader chain. While updating *OkHttp* from various versions to the newest one, four apps were reported as a crash because of a failure in finding `Platform` class in the path of the library update.

Source code analysis We discovered that `Platform` class in versions before 3.4.0-rc1 of *OkHttp* is named as `okhttp3.internal.Platform`, which conflicts with the one named `okhttp3.internal.platform.Platform` in version 3.9.0. From the exception stack, we know that those failed apps all include *OkHttp Logging Interceptor* (*okhttp3.logging.HttpLoggingInterceptor*) library, which is a sibling library of *OkHttp* and uses it as a dependency. As mentioned before, *LibScout* uses a root package matching to detect library invocations. That way, invocations between sibling libraries like *OkHttp Logging Interceptor* and *OkHttp*, whose method signatures start with the same root package, will be misreported as a library internal call. Thus, changes in interfaces exposed to sibling libraries will be missed by *LibScout*. Even finding such cases with auxiliary information besides the library API is hard, for instance, the *OkHttp* changelog for the whole okhttp family does not mention an internal `Platform` class renaming, since this class is not supposed to be invoked from outside this library family. To update the library based on API compatibility more effectively, a more fine-grained matching filter for sibling libraries and internal public interfaces should be applied to the lib usage detection logic of *LibScout*, which would very likely decrease the reported rate for updates to the max version. For example, in this case, 3 out of 4 apps could still be updated to the intermediate library version 3.3.1, the last version before `Platform` renaming.

```java
// version 4.18.0 November 30, 2016
public static synchronized void sdkInitialize(...){}

// version 4.19.0 January 25, 2017
@Deprecated
public static synchronized void sdkInitialize(...) {
...
// We should have an application id by now if not throw
if (Utility.isNullOrEmpty(applicationId)) {
throw new FacebookException("A valid Facebook app id must be set in the AndroidManifest.xml or set by calling FacebookSdk.setApplicationId before initializing the sdk.");
}
...
}

// version 4.26.0 August 24, 2017
// the same as 4.18.0
```

Listing 4.2: Evolution trace of sdkInitialize method

4.7.1.3 FacebookException

This exception is a custom exception that is thrown when an internal error happened in *Facebook SDK*. In our test set, 52 *Facebook SDK* failure apps reported an application id missing error during SDK initialization after update to version 4.26.0 (the newest *Facebook SDK* version in our repository) and the original library versions vary from 4.0.1 to 4.17.0. Thus, the *Facebook SDK* initialization must have changed with some version after 4.17.0. We look into the *Facebook SDK* upgrade guide and find a description about upgrading 4.18.0 to 4.19.0: *"The Facebook SDK is now auto-initialized on Application start. If you are using the Facebook SDK in the main process and don't need a callback on SDK initialization completion, you can now remove calls to FacebookSDK.sdkInitialize."*

Source code analysis To verify if this modification is the main reason of failures, we check the source code of *Facebook SDK* and discover that before version 4.19.0, the *Facebook SDK* is usually initialized manually via interface `FacebookSdk.sdkInitialize` (see Listing 4.2). The application id could be set either in `AndroidManifest.xml` file or `setApplicationId` method. The id could be set either before or after `sdkInitialize`. However, starting from version 4.19.0, interface `sdkInitialize` is labeled as deprecated, and now it is called by *Facebook SDK* automatically without explicit code invocation in host components. Deep within the initialization code, we find that the application id must be set before invoking `sdkInitialize` as shown in Listing 4.2 or otherwise an exception is thrown. Thus, the application id should be set as early as possible to avoid any failure. In fact, to support automatic initialization, *Facebook SDK* imported a new `ContentProvider` component `FacebookInitProvider` in 4.19.0. `ContentProvider` components can be initialized at the beginning of app launching ahead of any other components. By invoking `FacebookSdk.sdkInitialize` in `FacebookInitProvider`, the *Facebook SDK* can be initialized at a very early stage. In a standard *Facebook SDK* integration, `FacebookInitProvider` in *Facebook SDK*'s custom library `AndroidManifest.xml` file will be merged with the app's `AndroidManifest.xml` file during app building, and the application id should be

```
// file assets/www/js/services.js
facebookConnectPlugin.api('/me?fields=about,bio,
email,name,first_name,last_name&access_token=' + authResponse.accessToken, null, ↷
    ↪ ...);
```

Listing 4.3: Graph Request in SnapOdo

configured in `AndroidManifest.xml` file to ensure the application id is available during `FacebookInitProvider` initialization at app launching time. Changes to the `AndroidManifest.xml` are excluded from our test settings, and all the original library SDK configuration is kept as in the original app. Thus, some apps with lower library versions that set the application id after invoking `sdkInitialize` will fail with the newer library versions.

4.7.2 DroidMate Finding

To explore the incompatibility of libraries beyond crashes, we investigate the case for which we found a deviation in the runtime behavior in the *DroidMate* test after updating the *Facebook SDK* library. The *Facebook SDK* of app *SnapOdo* is updated from version 4.15.0 to the latest version 4.26.0 and after that failed to login to the facebook account. From the official changelog, we know that a Graph API upgrade occurred in version 4.16.1. According to the changelog of Graph API version 2.8, some deprecations happened, including the removal of a *"bio"* field on the `User` node. In Android apps, `GraphRequest` is usually created by either JavaScript or Java code integration with some fields defined in the graph path string. We decompile the *SnapOdo* package and find the `GraphRequest` creation in a JavaScript file as shown in Listing 4.3. The usage of the *"bio"* field is incompatible with the new Graph API used in newer *Facebook SDK* versions and leads to the login failure in this app. This case reflects potential updating obstacles beyond API-compatibility. For both integration options, field *"bio"* works just as a part of a string parameter that is definitely out of the range of *LibScout* detection.

4.7.3 Case Study

From those failure cases, we noticed that even though the APIs of different library versions are compatible, some internal execution logic changes could prohibit a simple drop-in update. We use the factors discovered in case of the Facebook exception as a case study and perform a large-scale analysis to evaluate the prevalence of such impeding factors for drop-in replacements in other libraries. It is worth noting that **1)** *Facebook SDK* labeled interfaces which are not recommended to use after some updates with a *"deprecated"* annotation instead of removing them directly, which puts them outside of *LibScout*'s API compatibility analysis; **2)** a drop-in update cannot change the configurations defined in `AndroidManifest.xml` file, which could be different between different versions.

Deprecated methods

We carried out a statistical analysis of the source code of 1430 different versions of 44 open source libraries that we gathered from maven repository.

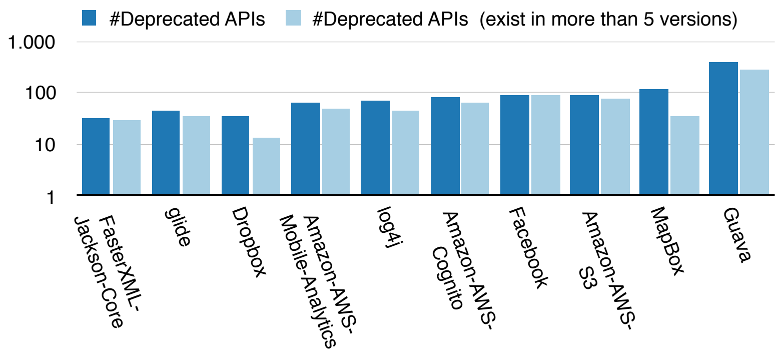


Figure 4.6: Number of public deprecated APIs in libraries source code and the number of them that exist in more than 5 versions

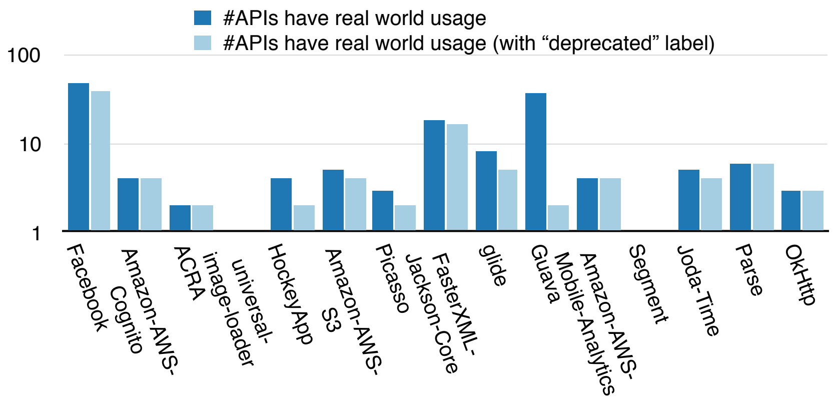


Figure 4.7: Number of public deprecated APIs that exist in more than 5 versions and that are used in apps (total vs. with "deprecated" label)

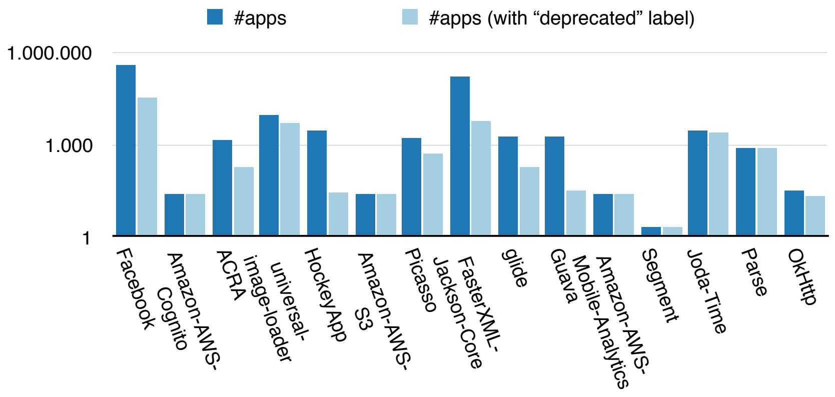


Figure 4.8: Number of apps that use public deprecated APIs (exist in more than 5 versions) and their usage (total vs. with "deprecated" label)

Table 4.3: Library manifest changes across different versions

Manifest Entries	#Changed Cases	#Library Concerned
Activities	16	ACRA, CleverTap, Facebook Audience, Facebook SDK, HockeyApp, Paypal, Braintree Payments, LeakCanary, Vkontakte
Services	7	ACRA, MapBox, Parse, Braintree Payments
Content Providers	2	ACRA, Facebook SDK
Broadcast Receivers	3	CleverTap, Vkontakte
Permissions	10	ACRA, CleverTap, Facebook Audience, HockeyApp, Parse, Paypal, Braintree Payments, LeakCanary

Table 4.4: Rules to identify incompatible updates when considering our discovered factors

Library	Side Effect	Original	Updated	Features
OkHttp	AbstractMethodError	= 3.0.0-rc1	> 3.0.0-rc1	Existing host implementation of `okhttp3.Callback`
	ClassNotFoundException	< 3.4.0-rc1	>= 3.4.0-rc1	Using library LoggingInterceptor together with OkHttp
Facebook SDK	FacebookException	< 4.19.0	>= 4.19.0	Invoking `sdkInitialize` without either invoking `setApplicationId` or defining applicationId in AndroidManifest.xml
	Login Failed	< 4.16.1	>= 4.16.1	Using field "bio" in graph requests

Table 4.5: Results of library updatability re-estimation

Library	#Apps	#Updatable		#Latest Updatable	
		LibScout	Re-Estimation	LibScout	Re-Estimation
OkHttp	104,046	97,176 (93.40%)	94,550 (90.87%)	45,962 (44.17%)	37,934 (36.46%)
Facebook SDK	199,007	187,191 (94.06%)	187,189 (94.06%)	145,817 (73.27%)	134,035 (67.35%)

We extend *javadocextractor*[7], which is a wrapper of *javaparser*[8], to check the occurrence of deprecated interfaces in libraries. We find that 32 of 44 (72.73%) libraries have deprecated methods. Among all those libraries with deprecated interfaces, 24 of them have deprecated interfaces present in more than 5 versions, which indicates the prevalence and permanence of deprecated methods. Figure 4.6 lists the deprecated API details for 10 libraries. To quantify the impact of those deprecated methods in real-world apps, we compared those deprecated interfaces that exist in more than 5 library versions with library invocation calls detected by *LibScout* from a more extensive app repository which contains 9,902,533 profiles for 2,041,017 apps. Since an interface is usually used before being deprecated, we also distinguished the usage situation for both non-deprecated versions and deprecated versions. Our results show that 20 of 24 libraries, 158 APIs in total, are detected as used in real-world apps. In those 20 libraries, 15 of them with 94 (59.49%) APIs in total, are used under deprecated status. Figure 4.7 shows the target API usage details and highlights the deprecated usage for 15 libraries. The amount of apps affected by deprecated APIs is also remarkable. In our results, 561,671 app profiles are reported containing target API calls, while 47,966 of them include those calls under deprecated status. Figure 4.8 lists the number of apps that include target APIs under deprecated status for 15 libraries. From the results above, we can see that most of the libraries have deprecated methods. A deprecated method is supposed to be removed in the near future, but based on our results, those methods usually remain for an extended period, which gives developers the chance to keep using outdated code and also brings false positives to API-compatible library updating. The prevalence of deprecated cases further shows that a plain drop-in replacement cannot work as good as expected.

Manifest changes Usually, library developers define necessary components and permissions in library manifest files which will be automatically merged with the app's manifest file when building the app with *Gradle*. This process could be opaque to app developers. In a drop-in replacement library updating, those manifest modifications, e.g., `FacebookInitProvider` registration in our test, will be ignored since no app rebuilding is performed. This can impede the library updating as we have discovered for the *Facebook SDK*. To gain insights on the extent of this problem, we gathered 362 Android Archive packages (i.e., manifest plus code) for 15 libraries and analyze the component and permission changes in manifest files across different versions. The result is shown in Table 4.3. Among all 15 libraries, 16 `Activity` changes happened in 9 libraries, 7 `Service` changes happened in 4 libraries, 2 `ContentProvider` changes in

[7]https://github.com/ftomassetti/javadoc-extractor
[8]https://github.com/javaparser/javaparser

2 libraries, 3 `BroadcastReceiver` changes in 2 libraries, and 10 permission changes in 8 libraries. In other words, 11 out of 15 libraries have at least one entry modified between versions. These frequent changes indicate a high potential for incompatible drop-in replacements despite API compatibility.

4.7.4 Library Updatability Re-Estimation

Our dynamic testing results reveal that failed library updates come from both flaws in the *LibScout* tool and library internal changes. Our case study confirms the prevalence of those factors across different libraries. The API-compatibility based updatability rate reported by *LibScout* should be adjusted. Here, we set *OkHttp* and *Facebook SDK* libraries as two typical examples and re-estimate the API-compatible based updatability rate after considering the discovered factors. We use the same app set as in our automated UI tests (332,432 apps in total). First, we gathered the theoretical API-compatible based updatability rate according to the compatibility definition of *LibScout* [53]. Then, we create rules to identify apps with incompatible library updates when considering our findings, as shown in Table 4.4. Lastly, we scan app profiles and filter out all the apps that match one of the rules. Method call information like `sdkInitialize` and `setApplicationId` is gathered by *LibScout* already, we only need to extend it with host interface implementation checking, manifest metadata (applicationId), and JavaScript analysis results (field *"bio"*). Considering field *"bio"* can be added to graph requests through not only JavaScript but also Java code, we take advantage of the *ARTist* [1] tool to filter any field *"bio"* usage in graph request construction relevant string flows. The final re-estimation results are shown in Table 4.5. We find that the updatability rate varies between 93.40% to 90.87% for *OkHttp* and stays (94.06%) for *Facebook SDK*. However, the updatability rate to the latest version varies more significantly between 44.17% and 36.46% for *OkHttp* and between 73.27% and 67.35% for *Facebook SDK*. The re-estimation result exhibits a decrease of the updatability rate compared to plain *LibScout*, in particular, the latest version updatability rate, when taking our discovered impeding factors into consideration. With runtime app behavior profiling, we find that a drop-in replacement for library updates is *technically* possible, but if a functioning continuous updating model is expected, the joint efforts from library developers, app developers, and *LibScout* tool developers are necessary to address those factors.

4.8 Discussion

We discuss the limitations and prospects of our study.

4.8.1 Research Sample

We used three libraries from different categories for our study. Although those are popular libraries, their results might not generalize and cover all kind of potential problems. However, our work still revealed important issues of library updates and shows that API-compatibility alone is not a good indicator for library updates. Further, we investigated 1.4k other library versions and 2M real word apps for identical problems and could confirm the prevalence of those problems, which we think makes them representative. Moreover, scaling the analysis to larger-scale and more intricate problems is naturally limited by the small-scaling of dynamic testing. Future work could investigate certain problem classes in a focused way.

4.8.2 Entangled Dependencies

A crucial observation of our tests is entangled dependencies between different libraries and even the host app. For instance Figure 4.9: both the app and the library L_a depend on library L_b. When updating L_a, not only $Call_{ha}$ but also $Call_{ab}$ should be taken into consideration. A more complicated case is that the host application creates an object from secondary dependency L_b and passes it to L_a as a parameter. It is not supported by our classloader customization based test framework. API static analysis result from our test samples shows that 1.7% of library APIs could be affected by this problem and also two failure cases in the unknown crashes of the dynamic test are confirmed to be related to this problem. This exceptional case is the limitation of our framework setting, but we here only focus on incompatibility cases brought by library updating. The crash cases reported in Section 4.7 are not affected by this exceptional case. Apart from that, our framework ensures that all dependencies are correct for host app and updated library respectively since the original library and its dependencies are still in the app. Obviously, the numbers reported in previous Section 4.7.4 are an *optimistic* estimate when no direct dependency conflicts occur. The situation for entangled dependencies in real-world might be far from desirable. We looked into the impact of dependencies on library updatability. We crawled library dependency information from Maven Repository[9] and limit a library's possible update to only versions that share the same dependency set with the original one. Compared to a purely API-compatibility based update, this API/dependency-based update shrinks the updatability rate significantly. The rate for the same 332,432 apps reduces by 47.95% (93.40%-45.45%) for *OkHttp*, 40.37% (94.06%-53.69%) for *Facebook SDK*, and 36.38% (99.94%-63.56%) for *Facebook Audience*. This multiple dependency situation is not a corner case. Static analysis of those apps shows that 57.50% of the apps that integrate *OkHttp* have invocations to *OkHttp*'s dependencies in either their host code or *other* libraries and even 96.03% for *Facebook SDK* and 97.76% for *Facebook Audience*. Hence, whether a lib can be updated in reality might also be constrained by further dependencies by other libs or app code to its own dependencies and, hence, in case of a conflict, prevent an update of the target lib without doing extensive updates of other libraries (potentially creating a *"dependency hell"*).

4.8.3 Framework and UI-based Testing Limitations

Although very carefully designed to avoid errors/crashes of the apps and libraries due to erroneous drop-in replacements, we cannot entirely exclude that some of the crashes of apps come from our framework, since it is unrealistic to debug all the failed apps from our testing. However, our investigation focused on those crashes with clear

[9]https://mvnrepository.com/

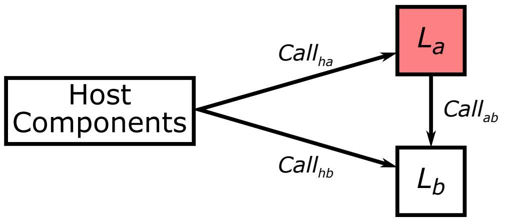


Figure 4.9: An example of entangled dependencies inside an app

problems stemming from the library integration and internal changes. Further, we only test control flows starting at `Activities` and achieve with this on average 35% app block coverage. Thus, our results form a lower bound on the potential problems of the tested libraries. The emphasis of our work is on confirming the existence of API-compatibility based update problems and identifying advice for future library update tool developers/researchers about what impedes library updatability.

4.8.4 Efforts from Multiple Parties

The main idea of this work is evaluating ways of (supporting developers in) maintaining dependencies, starting with evaluating the feasibility of drop-in updates and discussing the relevance of our results for library updatability. Our discovered problems are intricate, and hence any support for automatic lib updates or even tools that help developers in making a judgment of the library updatability have to consider those non-trivial problems, e.g., clear connections to changelogs, changed data structures, or code annotations. Multiple parties are involved in the library update chain and there is a call for action to better support lib updates in the mobile ecosystem, including better tools for app developers to judge and realize library updates or a call to system vendors to rethink the static linking of libraries in favor of more dynamic approaches (e.g., on Linux) that not only can profit compartmentalization of third-party code [26] but also its updatability.

4.8.5 Updating in Automated App Testing

In our *DroidMate* test, we observed a case of a highly non-deterministic app that resulted in exploration noise. The reason for the non-determinism is that the app has a lot of random actions, for example, loading different advertisements in different runs. Considering that our update framework opts in library updates as a replacement of the original library without any actual app code modification, we plan to investigate the possibility of migrating our lightweight framework to blacklisting unwanted libraries in automatic app testing.

4.9 Conclusion

Outdated third-party libraries are prevalent in apps. To alleviate the unpleasant situation, prior work suggested an API-compatibility-based library update solution using drop-in replacements of outdated libraries. In this work, we study the library updatability using such drop-in updates. We implemented a library update framework for Android and used it on 3,000 real-world apps for 3 popular libraries. Using dynamic testing of those apps, gave us insights into the runtime behavior of an API-compatibility-based updating solution. To discover more intricate incompatibility cases, automated user interface testing was carried out on 15 apps both before and after library updates. Our tests revealed intricate factors that prevent a drop-in replacement of libraries. Studying the source code of libraries that failed to update and using static app analysis, we confirm the prevalence of those problems in other libraries. Our re-estimation of prior estimates of the library updatability rate under consideration of the discovered impeding factors shows a decrease in the rate by more than half due to entangled library dependencies. This work is the first to confirm the existence of API-compatibility-based update problems and can provide valuable insights for future library update tool developers/researchers on what should be taken into account when updating libraries.

5 Privacy-Enhanced Accessibility Framework

Constraining Accessibility Service Misuse on Android

5.1 Motivation

Chapter 3 and Chapter 4 present our mitigation solutions against privacy and security issues from untrusted third-party libraries. In this chapter, we draw attention to Android's accessibility services, which are another type of untrusted component frequently included in applications. Accessibility features, also known as *a11y services*[1], are meant to assist people with disabilities or impairment using their computer systems. Since Android version 1.6, Android has included an accessibility framework that allows authorized third-party apps to act as accessibility apps, such as screen reader apps or alternative navigation apps via voice commands and head gestures. As accessibility apps necessarily have to be exempted to a certain extent from the usual isolation between apps, Android provides a dedicated permission `BIND_ACCESSIBILITY_SERVICE` to restrict access to the accessibility framework. Applications can retrieve information from the accessibility framework about other apps, or send events to other apps (e.g., UI interactions) only if they are explicitly authorized by the user as described in Section 2.1.2. However, this permission is coarse-grained and very powerful. Once an app has been granted access to the accessibility framework, it has the privilege to access private data from all other apps, including sensitive data normally protected by other permissions, or to mimic users' actions (like button clicks). According to Google's guidelines, the accessibility features are supposed to be used only by accessibility apps that help disabled and impaired users to operate their devices and apps. Despite this guideline, there exist a lot of apps that use these powerful features for their own purposes, for example, automatization of tedious user actions (e.g., easy uninstallation of apps via injected button clicks that navigate the *Settings* app or auto-filling of credentials by password manager apps). Given the power of accessibility apps and the widespread usage of accessibility features, it is reasonable to assume that not all apps use this power appropriately [85] and that the current accessibility framework has potential threats to user privacy. Even worse, recent reports have confirmed the existence of various malicious applications [86, 87, 88] that utilize accessibility features to monitor and mimic user interactions with third-party applications in order to steal sensitive data, such as user credentials or banking information, while the risks of accessibility features have been emphasized in academic works [89, 90]. What should be clear by today is that the current restrictions to access the accessibility framework are not sufficient to protect user data and defend against malicious intents.

5.2 Problem Description

Security and privacy concerns from accessibility frameworks Already in 2013, Kraunelis et al. [91] demonstrated a malware that utilizes Android's accessibility framework. Jang et al. [89] studied the security of assistive technologies and identified multiple vulnerabilities on four popular platforms. Their result shows that the trade-off between

[1]a11y is the abbreviation of *accessibility*.

security, compatibility, and usability is the root cause of these vulnerabilities. Kalysch et al. [92] assessed the weakness of accessibility features and proposed corresponding *developer side* countermeasures. Diao et al. [85] evaluated Android's accessibility APIs with an analysis of the framework as well as a large-scale app analysis. Their result reveals the intrinsic shortcomings in Android's current design and confirms the broad misuse of the accessibility APIs. Fratantonio et al. [90] presented attacks when combining Android's `SYSTEM_ALERT_WINDOW` and `BIND_ACCESSIBILITY_SERVICE` permissions, thus, further highlighting the shortcomings in privacy protection of the accessibility framework. Follow-up work [93] demonstrated the usage of these permissions by malicious apps. Those works highlight the existing privacy concerns in the accessibility framework, but did not create an appropriate defense.

Problem of the existing defenses against malicious accessibility apps The burden to establish any defense today rests on the shoulders of the app developers that might fall victim to misuse of accessibility features. App developers can pro-actively exempt components or UI elements of their apps from being monitored by the accessibility framework in an effort to protect sensitive data or prevent misuse of UI elements. Naseri et al. [94] proposed a *developer side defense* against eavesdropping through accessibility features. In their work, multiple tools are implemented to detect apps that are vulnerable to eavesdropping, to automatically fix discovered vulnerabilities, and to notify users of potential accessibility service misuse. Unfortunately, not only do many app developers abstain from those defenses [94], but even worse, those defenses defeat the very purpose of the accessibility services. For example, an app developer of a mobile banking app that exempts the input field for the account number to avoid leakage via accessibility services would also exclude screen readers or voice command apps from reading back or writing that number. What is needed to not make accessibility and privacy mutually exclusive is an accessibility framework that supports a more fine-grained control over how its features can be used.

5.3 Contributions

In this work, we propose an extension to Android's default accessibility framework that enables the configuration of more fine-grained control over how accessibility features are used by accessibility apps. We start by investigating the integration and usage of the accessibility framework in 95 real-world apps that are either benign a11y apps, apps repurposing a11y features (e.g., automatization), or malware abusing accessibility features in order to better understand what kind of policy enforcement such a solution has to provide and which potential limits exist. Our results exhibit a clear tendency of how malware is currently misusing the accessibility features. However, our results also raise the challenge that malicious behavior and benign behavior are not distinguishable at the API boundary (e.g., which accessibility data and features are being accessed) and that a suitable solution has to control the data flows within accessibility apps.

Noticing parallels between our setting and that of IoT and augmented reality apps, we take inspiration from the ideas of *data processing pipelines* for AR apps [95] and of *quarantined code modules* with *opaque data handles* for IoT apps [96]. Transferring those ideas to our problem setting for a11y, accessibility apps access certain information from the framework and process them in a particular way, or they trigger certain accessibility actions as reaction to certain triggers. For example, a screen reader accesses text information and outputs an audio stream, or a virtual mouse app tracks eye movement and clicks buttons. The key idea of our solution is to make the single steps in such processing pipelines explicit and sandbox them in least-privileged service components. Accessibility apps then build their pipelines by chaining those services together and orchestrate their interactions. We enforce policies at their input/output boundaries to govern to which data and features each module has access. By keeping the overall pipeline in mind, those policies control how data can propagate within a single pipeline—sources to sinks—or under which circumstances a pipeline can trigger (accessibility) actions.

Although our study of existing malicious and a11y apps indicates that a policy that universally maximizes functionality for benign apps while simultaneously eliminating the potential for misuse seems unlikely, our solution allows configuration of a trade-off between functionality and protection according to users' needs (e.g., disabling accessibility features that are not necessary for the desired a11y apps). This is a clear benefit over stock Android's all-or-nothing protection against misuse of the accessibility framework. Since our design only changes the public APIs of the default accessibility framework (e.g., apps needs to register and orchestrate their modules), only developers of a11y apps need to adapt their code to the new setting but no other app developers are affected. We demonstrate this by porting two open-source accessibility apps to our enhanced accessibility framework. We make the following contributions:

Systematization of accessibility service integration We study the actual usage of accessibility features in real-world benign, utility[2], and malicious apps. Our results reveal patterns and behaviors how the accessibility API is misused. We believe those results contribute to a deeper understanding of how a11y features are being (mis-)used and can help future work in creating better defenses against a11y attacks.

Privacy-enhanced accessibility framework Based on the results of our systematization, we propose a privacy-enhanced accessibility framework. Privacy here means that data retrieved via the accessibility framework should not leak without authorization and that all accessibility actions should be authorized or triggered by the user or at most be inefficiently misused. Our framework separates a11y logic of apps into sandboxed code modules and allows enforcement of privacy policies at the input/output boundary of those modules. This enables a more fine-grained control over how accessibility features are used, how data propagates in the pipelines formed by those modules, and, hence, offers a more effective protection against misuse of the accessibility framework than stock Android.

Real-world app migration and evaluation We migrate two real-world open-source accessibility apps to our privacy-enhanced framework to demonstrate how our framework

[2]We refer in the context of this paper to apps that repurpose the a11y features for user desired but by Google unintended use-cases as *utility* apps.

provides better protection in those cases. Further, micro-benchmarks show that the performance overhead imposed by our solution is acceptable.

5.4 Study of Accessibility Service Usage

Considering the high privileges of an accessibility service and the diverse ways to use it—for a11y as intended, as a user-desired utility, or for malevolent purposes—we are interested in how real-world apps make use of this service and whether there exist distinguishing features in the usage patterns between a11y, utility, and malicious apps. Prior work [85] evaluated the usage of accessibility services in *normal*[3] apps based on natural language processing of the app descriptions. This approach highly relies on the accurate (and honest) developer documentation. A missing, ambiguous, or dishonest description could hide the actual usage of the accessibility features from the results.

To gain a more comprehensive and reliable understanding of the usage of accessibility features, we base our study of accessibility (mis-)usage directly on the apps' code, including utility and malicious samples. By collecting each sample app's access to the accessibility framework and then comparing the integration between each app's components and their accessibility services, we discover patterns how accessibility apps actually make use of the a11y framework and we gain an overview how accessibility can undermine the users' privacy in practice. We look at the different ways how an `AccessibilityService` is configured (e.g., which events are being subscribed), which APIs are being used, and which behavioral patterns can be detected in accessibility apps in the remainder of this section. The key question we want to answer is *if the different types of accessibility apps—a11y, utility, and malicious—are distinguishable in their configuration, API access, or use of accessibility services?*

5.4.1 Accessibility App Sample Set

While gathering app samples, we differentiate between three classes of accessibility apps: *malicious*, *utility*, and *a11y*.

Malicious apps take advantage of accessibility service to attack users, e.g., logging sensitive user input, mounting phishing attacks, stealing private data from other apps, or surreptitiously granting permissions and installing apps. To collect a representative and timely set of malicious apps for our investigation, we turn to renown malware repositories on GitHub. From GitHub, we collected 608 reported Android malware samples from top ranking malware repositories [97, 98, 99, 100]. After filtering out samples without an `AccessibilityService`, we obtained 55 *malicious* accessibility app samples (57 `AccessibilityService` implementations).

In contrast, *a11y* apps use specific accessibility features to assist people with disabilities or impairments. The use of the accessibility service in those apps meets the intended purpose by Google. For example, a screen reader app reads aloud the text label on a touched button to assist users with visual impairments in using the device. By keyword search on Google Play Store, we gathered 5 *a11y* sample apps. Lastly, *utility* apps are neither typical assistive apps nor malicious. They ignore Google's

[3]Here *normal* refers to apps in official markets.

Table 5.1: Accessibility service configuration in sample apps

Attribute	#Malicious (57)		#Utility (36)		#A11y (8)	
events from all apps[1]	49	(86%)	24	(67%)	8	(100%)
canRetrieveWindowContent	42	(74%)	30	(83%)	6	(75%)
$\cup$	57	(100%)	34	(94%)	8	(100%)
$\cap$	34	(60%)	20	(56%)	6	(75%)

[1] Service does **not** define an allowlist of package names

Table 5.2: Allowlisted package names in service configurations (no a11y app configured an allowlist)

Package	#Malicious (8)	#Utility (12)
com.android.settings	0	5
com.android.packageinstaller	0	2
browser*	0	3
communication*	2	1
shopping*	2	0
transportation*	2	0
tool*	2	0
self*	6	3

* Category of apps (since multiple packages of this type are monitored)

accessibility developer guide [12] by using accessibility features for user-desired functionality beyond supporting people with disabilities, such as optimizing user experience (e.g., automatization of tedious tasks or password auto-fill). Google once announced to remove apps that use accessibility features for purposes other than the intended way [101], but this ban was paused after Google realized the popularity of accessibility features in supporting non-accessibility functionality. We crawled 2,751 top Google Play Store apps in December 2018 and found 36 accessibility apps of this kind, which we use as our *utility* app samples. To check that both the *utility* and *a11y* apps are not malware in disguise, we scan those two sample sets with VirusTotal [102].One app, *Avira*, was reported as malware by VirusTotal. Considering it was flagged by only 1 of 60 engines, we conservatively removed it from our set but did not think this significant enough to report to Google Play Store. Our non-malicious app sets finally consist of 5 *a11y* apps with 8 `AccessibilityService` and 35 *utility* apps with 36 `AccessibilityService`.

In total, we collected 95 accessibility app samples (101 `AccessibilityService` implementations) for our investigation.

5.4.2 Accessibility Service Configuration

As introduced in Section 2.1.2, app developers can control the capabilities and types of events that their `AccessibilityService` will receive by customizing the `AccessibilityServiceInfo` configuration. This configuration provides a statement about which sensitive data from other apps is potentially exposed to the `AccessibilityService` via the accessibility framework. Among the different available configuration attributes, `packageNames` and `canRetrieveWindowContent` effectively constrain the accessibility app's access to other apps. Attribute `packageNames` allow-lists the source packages for `AccessibilityEvents` the `AccessibilityService` will receive. If this attribute is not set, the `AccessibilityService` will receive events from *all* other packages. If developed with least privilege principle in mind and if applicable, the `AccessibilityService` should specify all the necessary source app packages here. The attribute `canRetrieveWindowContent` controls if the accessibility app can access the window content of other apps, including sensitive data contained within those windows. Obviously, this access to window content is a great way to steal data.

We compare the accessibility service configurations for those two highly sensitive attributes within our app samples to understand the extent of sensitive data to which different accessibility apps have access to. Since this configuration can be set both statically and dynamically, we extract the static configuration file from the apps and combine this with runtime information from tracing the `setServiceInfo` system API. Table 5.1 shows the number of packages that do **not** declare a package name (i.e., monitor broadly) and that are able to inspect the window content of other apps. The results show that all malicious and `a11y` apps in our sample set monitor broadly, i.e., every malicious and `a11y` app is at least able to inspect window content or receive events of all other apps, while 34 (60%) of the malicious and 6 (75%) of the `a11y` services can do both. While this is intuitive, given the nature of those apps, also 34 (94%) of the utility services make use of those features, where 20 (56%) utility services use both features. For those services that specified an allowlist of package names, we also check the package name details. Of all malicious apps, 8 services set an allowlist and receive only events from listed packages, while 12 utility services set an allowlist. None of the 8 `a11y` services set an allowlist and all of them monitor broadly. The distribution of the (types of) allow-listed packages by the malicious and utility apps can be found in Table 5.2. Those results show that while utility apps listen primarily to events from system apps, like settings, installer, or browser, malware targets specifically packages in certain categories, such as communication or shopping.

Summary From those results, we conclude that the currently available constraints on accessibility service do not prevent the risk of abuse of `a11y` features, since all app types, including legitimate accessibility apps, configure a broad monitoring. Further, the similarity between the configurations makes it hard to distinguish purely on the configurations between a targeted attack and compliance to the least privilege principle.

5.4.3 Accessibility API Usage

Since the accessibility service configuration does not show a distinguishable pattern between different app types, we further investigated the accessibility framework API usage within accessibility services. After a review of the accessibility framework documentation, we categorize the accessibility API into three categories: **1)** *retrieve information*, **2)** *perform node action*, and **3)** *perform global action*. *Retrieve information* APIs refer to interfaces that request information about other apps, including on-screen text, window position and so on. *Perform node action* API refers to interfaces that perform an action on a specified UI element (*node*), e.g., clicking a button. *Perform global action* API refers to interfaces for issuing a global operation, like clicking the *"home"* button or showing the recent task list. Based on this categorization, we analyzed the types of accessibility APIs that are used in our sample apps and with which goal they were used by the apps (i.e., scenario). To this end, we manually interacted with the app UI and pinpointed possible usages based on the service descriptions and hints of UI elements. Since malicious apps by nature might mislead the user in those descriptions, we further collected accessibility-related behavior descriptions from technical reports by malware analysts and reverse engineers. For each discovered usage scenario, we manually inspected one app in depth through either reverse engineering or source code analysis where possible to find patterns how accessibility services are integrated into their apps. In the end, we found four common patterns for the usage of accessibility methods:

Pattern P1: retrieve information $\Longrightarrow$ accessibility app operation The accessibility apps digest the retrieved information about other apps locally, but do not trigger any global/local accessibility action. For example, a screen reader app gathers screen texts and then processes this information in a separate `TextToSpeech` component to read it aloud.

Pattern P2: retrieve information $\Longrightarrow$ node action Here, first a node is selected based on information retrieved from the accessibility framework (e.g., locating a specific button) and then an action is triggered on that specific node (e.g., clicking). For instance, a facial access app that allows controlling the device via facial and head gestures can perform a click on a button to which the users points with such a gesture.

Pattern P3: retrieve information $\Longrightarrow$ global action Different from pattern P2, information gathered from the accessibility framework about another app is used to trigger a global action. One typical scenario is a switch access app that captures the *"home"* key event from an external keyboard and then performs the global action to go back to the home screen.

Pattern P4: accessibility app operation $\Longrightarrow$ global action In this pattern, the app triggers a global action purely based on app-internal results but without first retrieving any information about other apps from the accessibility framework. For instance, a soft key mapping is one example for this pattern.

Table 5.3: Patterns of accessibility API usage

	Scenario	**Patterns**			
		P1	**P2**	**P3**	**P4**
Malicious	Content Eavesdropping	✓	✗	✗	✗
	Phishing	✓	✗	✗	✗
	Process Persistence	✓	✗	✗	✗
	Silent Installation	✗	✓	✓	✗
	Silent Privilege Elevation	✗	✓	✓	✗
	E-Banking Fraud	✗	✓	✓	✗
Utility	Fingerprint Gesture	✓	✗	✗	✗
	App Locker	✓	✗	✗	✗
	App Usage Tracing	✓	✗	✗	✗
	Browser Usage Tracing	✓	✗	✗	✗
	TextView Mapping	✓	✗	✗	✗
	Notification Replay	✓	✗	✗	✗
	Smart Reply	✓	✗	✗	✗
	Auto Permission Grant	✗	✓	✗	✗
	Password Auto Fill	✗	✓	✗	✗
	Web Control	✗	✓	✓	✗
	(Un)Installation Protection	✓	✗	✗	✗
	Auto Uninstallation	✗	✓	✗	✗
	Deep Clean	✗	✓	✓	✗
	Battery Save	✗	✓	✗	✗
	Global Menu	✗	✗	✗	✓
A11y	Screen Reader	✓	✗	✗	✗
	Speech to Text	✓	✗	✗	✗
	Facial Access	✗	✓	✓	✗
	Gesture Access	✗	✓	✓	✗
	Voice Access	✗	✓	✓	✗
	Switch Access	✗	✓	✓	✗

✓: uses pattern ✗: does not use pattern

Summary Table 5.3 shows the mapping between usage scenarios and the integration patterns for different types of apps. From those results, we can see that scenarios from different categories can have the same API integration pattern. For instance, silent app installation, deep clean, and voice access share patterns P2 and P3. This makes a static detection of accessibility API misuse based on the integration pattern infeasible. Thus, also heuristics based on which APIs are being used—a common technique for malware detection—cannot sufficiently distinguish the different app types purely based on the observed API usage patterns.

Table 5.4: Accessibility pipelines for different app types and scenarios

	Scenario	**Trigger**	**Intention**
Malicious	Content Eavesdropping	Auto enabled	Send to remote
	Phishing	Target app operation	Load a phishing page
	Process Persistence	Target app operation	Back home
	Silent Installation	Ad click	Click specific buttons in specific app
	Silent Privilege Elevation	Auto enabled	Click specific buttons in specific app
	E-Banking Fraud	Auto enabled	Text input & click specific button in specific app
A11y	Screen Reader	Finger select	Read text aloud
	Speech to Text	Auto enabled	Enable shortcut button
	Facial Access	Camera detection	Screen navigation
	Gesture Access	Finger gesture	Screen navigation
	Voice Access	Microphone detection	Screen navigation & text editing
	Switch Access	Hardware keyboard	Screen navigation & text editing

5.4.4 Complete Accessibility Pipelines

Table 5.3 shows the high-level API-based patterns for interacting with the accessibility framework, which contain both retrieving data (patterns P1, P2, and P3) and triggering actions (patterns P2, P3, and P4). Since different app types cannot be distinguished at that abstract level, we now take app-specific contexts around those patterns into consideration and zoom in to apps to investigate the various events that trigger access to the accessibility framework, how data retrieved from the accessibility framework is used, and to which sinks such data flows. For simplicity, we call those app-specific combinations of triggers and usage the apps' *accessibility pipelines*. By comparing the pipelines of malicious applications and benign applications of the accessibility framework, we can pinpoint further similarities and differences between different app categories. The results of investigating the accessibility pipelines for different app types and scenarios are summarized in Table 5.4. We explain this table in the following, when we discuss the similarities and differences between malicious apps and a11y apps after comparing their triggers and intentions.

Similarities **1)** Although the triggers of the two app categories vary a lot, the commonality is that all apps determine trigger events themselves. Here, *target app operation* means that an app that is monitored with the help of the accessibility framework performs a specific operation (e.g., comes to foreground on screen), while in the remaining triggers the accessibility app reacts to specific stimuli from the user (e.g., finger or facial gestures) or it reacts to arbitrary custom logic (e.g., auto-start when service is registered). In any case, evaluation whether a trigger condition is met resides entirely within the apps. **2)** We found that 2 out of 4 prominent intended operations in a11y apps overlap with the intended operations in malicious apps: voice access provides voice controlled text editing support, which overlaps with the text input in malware that mimics user interactions in e-banking fraud; and facial access provides screen navigation through a camera-based mouse that performs button clicks, which are also used by malware for, e.g., silent package installation and granting permissions. **3)** Although the intentions of screen reader, voice access, and content eavesdropping are not the same, all of them require raw data processing within the app. Hence, the raw data usage is opaque without precise data flow analysis and constraints. This also affects utility apps. For instance, *McAfee Safe Family* transmits user web and app usage tracking data to their server to support multi-device parental control—behavior that uses the accessibility framework similarly to content eavesdropping malware.

Differences **1)** We noticed that although some apps from different categories share the same intentions, a11y apps usually require more powerful accessibility functions. For example, the silent installation scenario requires clicking specific buttons in the settings app, while facial access supports users in clicking any button in any app on screen for navigation. That means, in fact, malicious apps can be easily over-privileged without raising immediate suspicion. **2)** Both benign and malicious apps require raw content processing within other components of the apps, but their final data destinations are different. For example, we found audio as data sink for screen reader, UI as the destination for voice access text editing, and network interface as sink for malicious content eavesdropping. **3)** By comparing the triggers of the pipelines, we found that malicious apps are more likely to perform operations silently or against users' intentions. Three of the malicious pipelines are auto enabled after accessibility service activation. No user involvement is needed. The other three triggers react to specific user operations on itself or third-party apps (similar to `a11y` apps), however, the reaction violates the users' expectations (e.g., a phishing page is shown). In contrast, triggers in benign apps are more likely to be user-explicit and the corresponding reactions are always in conformity with user intentions. For example, switch access clicks the same buttons as silent installation, but this click action is explicitly triggered by the user through a keyboard press.

Summary The fact that `a11y` apps need a more general access to accessibility features (e.g., being able to press any button in any app) prevents a simple least-privilege policy on access to the accessibility framework in order to constrain misuse of accessibility features. Further, the comparison shows that the pipelines of different app categories share similar triggers and actions, thus, like API patterns (see Section 5.4.3), differentiation of app types purely on only concrete triggers or concrete actions is not feasible. The crucial difference between the app categories that we find is that driven by the category of the app, the complete pipeline is distinguishable when being able to detect the combination of which trigger lead to which action or data leak. For instance, a screen reader has full access to all screen content but only needs the audio API as a data sink to read discovered texts and labels. Or, a facial access app needs to click an arbitrary position that was determined from the user's head movement in the camera feed. Unfortunately, all apps evaluate their trigger conditions themselves and the accessibility pipelines in stock Android are opaque to any fine-grained enforcement of control and data flows. This leads us to our key insight for our solutions.

5.5 Key Idea and Threat Model

From our study, we learned that benign accessibility apps distinguished themselves from malicious ones through different data destinations in combination with explicit user-consented node actions, both of which are dependent on the purpose of the `a11y` app. Benign accessibility apps usually gather user intentions through either on-device sensors or peripherals. Then they take advantage of accessibility features to either perform specified UI operation based on user intention (e.g., clicking a button) or collect necessary user-requested information from the application framework and other apps. These gathered sensitive information may finally be consumed by components that provide feedback to the user (e.g., audio output). Thus, we define *privacy* in the context of our work as *data retrieved via the accessibility framework should not leak without user authorization and all node actions should be authorized or triggered by the user or at most be inefficiently misused.*

In consideration of those insights, a potential privacy-enhanced accessibility framework should **1)** associate the UI operations by an `AccessibilityService` with user intentions to avoid (covert) malicious node actions or at least withhold crucial information for efficient, malevolent node actions; and **2)** prevent the on-screen information of apps that is gathered by an `AccessibilityService` from being misused by malicious accessibility apps (e.g., unauthorized leakage of sensitive information). To illustrate, consider Figures 5.1 and 5.2. The accessibility app in Figure 5.1 acting as a supposed screen reader can consume textual information from the accessibility framework as input and can write arbitrary output streams to audio sinks. To avoid the screen information from leaking or the app from issuing malicious node actions, it should not be allowed to issue node clicks or use any other output channel (i.e., least-privilege). Thus, it can work as intended as a screen reader while preventing sensitive data leakage or surreptitious interactions with other apps. Similarly, the accessibility service in Figure 5.2, acting as a facial access app, can receive arbitrary input from other components of the accessibility app (e.g., results of processing video data or motion sense API for gesture recognition), but should only issue clicks *"blindly"* to certain UI elements or global events. That means the coordinates for button clicks should come from the video processing component, which in turn can only consume camera feeds, but the app should not be able to analyze the screen content otherwise. Then, since then the app cannot efficiently explore other apps' UI since it lacks feedback about screen hierarchy, misusing accessibility features for maliciously installing apps or granting permissions is impeded.

Key idea

The key idea of our solution, whose implementation we present in the following Section 5.6, is **1)** to treat the accessibility pipeline of accessibility apps as a sequence of steps, such as trigger detection, local processing, and output streams or node actions; and **2)** to redesign the accessibility framework such that those steps are made explicit and each step's privileges and I/O can be individually governed by a least-privilege privacy policy, however, the content of each step is treated as a blackbox. By keeping the overall pipeline in mind when authoring the privacy policy, we establish a control over the possible data flows of accessibility apps. With a suitable policy that allows benign flows to proceed while preventing potentially malicious flows, privacy protection and enabling accessibility services do not need to be mutually exclusive anymore.

Threat model

We assume that an accessibility app is malicious, meaning that all code, even when divided into individual sandboxed steps, can *collaboratively* still be malicious and steps be tailored to each other to use their individual access rights and I/O to implement an attack (i.e., unauthorized action via the accessibility framework or leakage of data obtained via the accessibility framework). Picking up the example in Figure 5.2, although the video processing component cannot inspect the screen content anymore to detect buttons, it could send hard-coded coordinates for click events that are independent of the camera feed in order to trigger clicks at coordinates desired by the attacker. We discuss the efficiency of our solution under this threat model in Section 5.8.

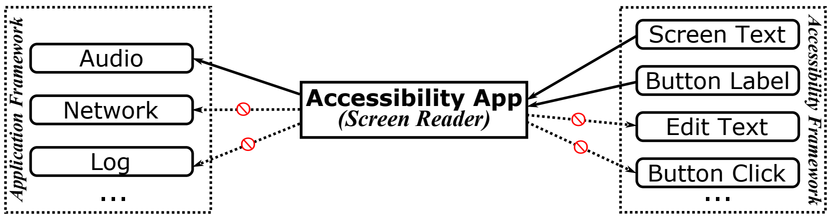


Figure 5.1: Example sandboxing for Screen Reader

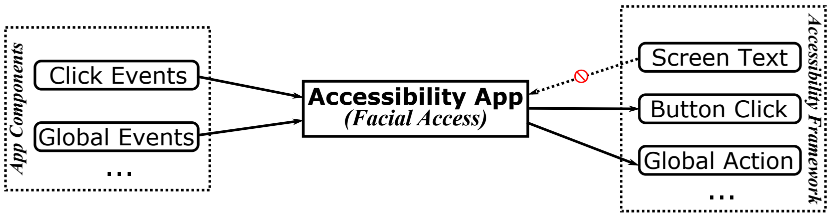


Figure 5.2: Example sandboxing for Facial Access

5.6 Privacy-Enhanced Accessibility Framework

In the following, we present the design concepts and implementation to realize our idea for constraining misuse of the accessibility framework.

5.6.1 Overview and Design Concepts

The key idea in our solution is that we treat the accessibility pipeline as a sequence of connected, individual steps and apply flow constrains to control the data flows along the pipeline to prevent unauthorized data leaks or actions. We categorize the steps of those pipelines into three types of code modules that are chained (see Figure 5.3): a *frontend module (optional)* to gather user intentions (e.g., from sensors or peripherals), an *accessibility module* to perform UI operations or retrieve sensitive information via the accessibility framework, and a *backend module (optional)* that creates the output of the pipeline (e.g., audio or text output). Those types have been directly derived from our previous observations about how a11y apps operate and we find them sufficient to implement the accessibility pipelines with minimum-function, least-privilege steps. Although such a logical pipeline exists in real accessibility apps, clearly distinguishable modules do not necessarily exist in current apps and their logic is commonly mixed together in app components. We demonstrate in Section 5.7 how accessibility apps can be retrofitted to our solution. To prevent misuse of the accessibility API in this pipeline, we transfer design concepts for privilege separation and information flow control to our solution. In particular, we found strong parallels to *opacified computation* for IoT apps [96] and to *recognizers* in augmented reality data processing pipelines [95].

Privilege separation We implement privilege separation of the involved modules, a common practice in other areas of privacy protection on Android, such as constraining third party libraries [27, 26, P1]. By default, all code running within the same app

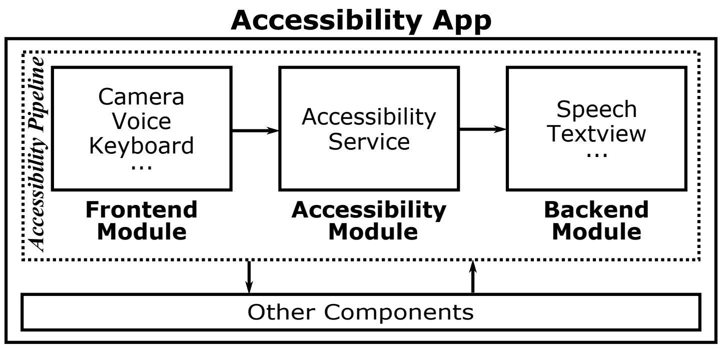


Figure 5.3: Accessibility pipeline with sandboxed modules

sandbox (i.e., under the same id in Android) would share the same privileges. Thus, to privilege-separate untrusted code, it is moved into a separate sandbox in form of another id under which it executes with a distinct set of permissions and access rights. This establishes a clear boundary between sandboxed (or *quarantined* [96]) code modules and allows access control at the process boundaries. Further, it allows control over the interactions between modules that are in separate sandboxes. We transfer this idea to the accessibility framework by composing the pipeline of actual, distinguishable code modules in their own sandboxes. Thus, we can control to which resources or APIs each module has access and designating each module for a certain step in the pipeline makes the overall process more transparent. No module by itself should have enough privileges to conduct the malicious operation. For instance, if a backend module of a text-to-speech app has to produce audio output, we allow this module to only access Android's audio API but not leak any data to the filesystem, other modules, or network sockets.

Information flow control

To build the pipeline, modules must interact with each other in a coordinated fashion. For instance, an accessibility module could accept screen coordinates as input and will output the on-screen information of the *node* (reference to UI element) at this particular location, which another module might receive to operate on (e.g., read out text elements of the UI element). One way to build these pipelines would be with direct IPC connections between modules. However, this would necessitate that the I/O interfaces of modules are tailored to each other, which would make the setup inflexible (e.g., if a frontend module could provide data to several kinds of accessibility modules) at no apparent security benefit. Instead, in our design, components of the accessibility app that are outside the pipeline connect modules and orchestrate the pipeline (e.g., forward the data between modules), which only requires each module to expose their own IPC interfaces to those app components via newly introduced I/O functions.This creates the risk that private data can leak to the components of the accessibility app that are not sandboxed or that those components can modify or counterfeit data exchanged between modules. To solve this problem, we take inspiration from *opaque handles* [96]: hidden references to raw data that are associated with a taint label and that can only be dereferenced within a sandboxed module. By only releasing handles to the orchestrating components outside the pipeline we prevent leakage of potentially private data to code that is not sandboxed and protect the integrity of that data from modifications by code outside the pipeline. By tainting the handles with the tag of the code module that output the data and checking those taints when handles are given as input to another module, we can ensure the authenticity of the received data and, further, can enforce simple flow constraints that govern how the modules have to be chained together. Originally [96], the taint labels also propagate to the taint label sets of module sandboxes and are forwarded to outgoing handles. That was necessary, since multiple flows might converge at a module and the context of the sandbox and of its outgoing data need to be distinguishable. Our design is simpler, since we have only a single flow in the pipeline and hence do not need to keep taint sets on sandboxes. Moreover, in contrast to the original work, we noticed that in some pipelines non-privacy-critical data could be released to the host app to allow, for instance, customizations (see Section 5.7.2 for such a scenario). Thus, our policy supports specifying that raw data can be released to components outside the pipeline by dereferencing the handle. To ensure the integrity and origin of all data, our solution allows only handles to be passed as input arguments to other modules.

Recognizers Lastly, we borrow the concept of *recognizers* used to limit sensitive data sharing in augmented reality data processing pipelines [95]: in place of getting raw video data, augmented reality apps subscribe to the output of certain trustworthy video processors (e.g., object *recognizers*) and only receive the minimal amount of data necessary for their operation. We noticed a similar setting in accessibility apps. Accessibility apps can depend on a pre-processing of raw data from sensors or peripherals, e.g., the camera. For instance, a facial mouse detects with the camera the user's head movements and gestures, and maps this to screen coordinates and click events. This would be done in our solution in the frontend modules (see Section 5.7.2). Although our threat model assumes all modules can be malicious and the output of the frontend module is generally not trustworthy, it is not unreasonable to assume that also scenarios exist in which the frontend module could be pre-installed or be coming from a trusted source, similar to *recognizers* in AR data pipelines that move common pre-processing to trusted system-provided component. A crucial benefit of a trusted frontend module in accessibility pipelines is that it provides a trusted source for detecting user intentions. In our design, we use this concept of *recognizers* by recording the outputs of frontend modules and later comparing them against the parameters of node actions. If the frontend is a trusted *recognizer*, this allows verification of node actions and to link user intentions with node actions.

5.6.2 Implementation

Figure 5.4 gives an overview of our implemented solution. We extend Android with a new service `PASManagerService` and its corresponding UI application `PASServer`. `PASManagerService` is the core component in our implementation. It provides accessibility apps with a new set of APIs to orchestrate their accessibility pipelines. It works as a central accessibility event dispatcher that bridges between Android's original

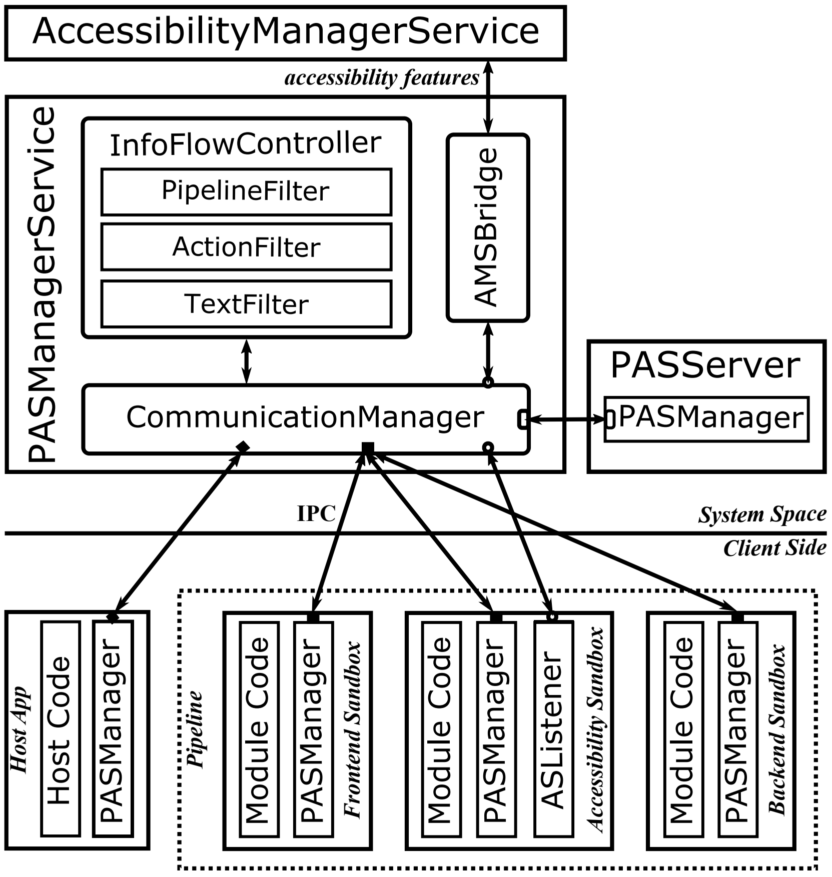


Figure 5.4: Privacy-enhanced Accessibility Framework

accessibility system service `AccessibilityManagerService` and the accessibility modules of client apps, i.e., client apps that want to make use of `a11y` features use our `PASManagerService` instead of the default `AccessibilityManagerService`. The `PASManagerService` itself is a system-side client (`AMSBridge`) to the `AccessibilityManagerService`. We implement the flow control for accessibility pipelines within the `InfoFlowController` of the `PASManagerService`. In the following, we will first introduce the details of the system-side `PASManagerService`, `PASServer`, and their components (Section 5.6.2.1). Then the new accessibility APIs and client-side integration will be introduced (Section 5.6.2.2).

5.6.2.1 System-Side Components

`PASManagerService` consists of three key components: `CommunicationManager`, `AMSBridge` and `InfoFlowController`.

CommunicationManager `CommunicationManager` is the IPC communication hub between host apps, the modules in their pipelines, the `AccessibilityManagerService` (via the `AMSBridge`), and a new settings app `PASServer`. We use the standard Android proxy-stub concept for *Binder* IPC, where every client to the `CommunicationManager` uses a `PASManager` to call the `CommunicationManager` and to receive callbacks from `CommunicationManager`. The host app and its modules also exchange data via the `CommunicationManager` with each other. If the host app and its modules could communicate directly with each other, this would necessitate that all *opaque handles* are set as part of the IPC communication (e.g., within *Binder*). This would be a very invasive change to a fundamental component of Android. By prohibiting direct communication between the host app and its modules as part of the modules' sandboxes, using a custom permission[4] unavailable to third party apps, and ensuring policies that prevent modules from leaking data to locations readable by the host app or other modules (e.g., SD card), we force modules and their host app to communicate via `CommunicationManager` with each other. Hence, while modules can be chained together, this solution ensures that they can only be chained through the `CommunicationManager` as a channel controlled by our framework. This places `CommunicationManager` in the position of a reference monitor to enforce information flow control (see further down).

AMSBridge Accessibility modules additionally need to communicate with the `AccessibilityManagerService` to make use of accessibility features. Instead of direct access to the `AccessibilityManagerService`, the `CommunicationManager` together with the `AMSBridge` bridges this communication. They provide to the accessibility module in a new manager class `ASListener` as much of the vanilla `AccessibilityManagerService` API as possible in order to reduce the effort of migrating apps from the original framework to our solution. In turn, `AMSBridge` is registered as an event listener to the original `AccessibilityManagerService` and dispatches these events to registered modules or forwards requested actions from the modules to the `AccessibilityManagerService`. This puts `AMSBridge` into a great position to enforce access control on the accessibility features used by modules.

InfoFlowController `InfoFlowController` realizes flow constraints on the communication passing through the `CommunicationManager`. It implements three types of flow constrains: `PipelineFilter`, `TextFilter` and `ActionFilter`.

PipelineFilter `PipelineFilter` implements the *opaque handles* and taint-based flow control. It maintains a set of unique identities for all modules as well as the host apps, and it keeps a mapping between module outputs and their handles and taints (i.e., identity of module that created the output). Thus, when a module sends an output to the host app, the data will be replaced by a new handle and the data be stored in `PipelineFilter`. The host app cannot use the handle to modify the referenced data. Every time the host app sends a handle to a module as input and the corresponding data is supposed to come from a specific other module as output, `PipelineFilter` uses the stored taint to validate this claim or otherwise abort the release of the data as input to the receiving module. Similarly, it releases raw data to the host app if the

[4]A future version of our solution could also use new SELinux types.

policy allows this and the host app requests dereferencing a handle. However, only a handle can be passed as input to another module, hence, integrity and origin of released data is always ensured when being further processed in the pipeline.

TextFilter `TextFilter` implements a similar control but for textual elements within `AccessibilityNodeInfos` returned from `AMSBridge` to modules. It replaces the plain text with a random uuid before sending the node info back to the host app. This uuid and plain text pair is stored in a map in `TextFilter` and only on input to an authorized module `TextFilter` releases this text. Thus, if `AccessibilityNodeInfos` is released to a module, `TextFilter` can decide whether that module is authorized to also receive textual content that could be privacy sensitive. Modules that are not in need of such information, e.g., because they only need the node for information about screen layout, can thus operate with lower-privileges. This can be easily applied to other content besides textual information, however, we have not encountered the need to hide other content yet.

ActionFilter `ActionFilter` validates the user intentions for action events when the frontend module is trusted, i.e., is a trusted recognizer component. `ActionFilter` records the output of frontend modules, e.g., the coordinate of a UI element that the user wants to click. Once `AMSBridge` receives a call to perform an action from a module, it asks `ActionFilter` to validate if the target UI conforms to the user intention recorded before. If the frontend module is trusted, a successful validation links the action to the user intention. Thus, a pipeline with a trusted frontend module is hindered in issuing actions that were not authorized (triggered) by the user. If the frontend module is not trusted, `ActionFilter` cannot help, since the frontend module and the accessibility module could be colluding to issue malicious actions.

PASServer Lastly, `PASServer` is a new settings app for our solution to assist users with accessibility feature management. Users can en-/disable a pipeline or (de-)activate the centralized accessibility service through this app.

5.6.2.2 Client-Side Integration

Modules are started by the `PASManagerService` very similarly to regular app sandboxes and their launch establishes a bi-directional communication between a module's process and the `PASManagerService`. When `PASManagerService` launches a module's application sandbox, it already receives a *Binder* reference to this process from Android, which allows `PASManagerService` to send messages to the module. After the module's application sandbox has been started and the module's code been loaded, it requests a *Binder* reference to the `PASManagerService`, which is encapsulated in a `PASManager` and allows it to send messages to `PASManagerService`. With this two *Binder* references a bi-directional communication is established. Modules that make use of accessibility features additionally register an `ASListener` with `PASManagerService` through which they can receive accessibility events and issue actions. The host app also has a `PASManager` that allows it to issue commands to `PASManagerService`, e.g., invoke modules and pass/receive data handles via `CommunicationManager`. For a full-fledged implementation, we envision that accessibility apps carry their modules as payload (separate *dex* files) and register them during installation in the privacy-enhanced framework, similar to how prior works proposed sandboxing third party libraries [27, P1]. Alternatively, modules could be provided as standalone packages on a market and accessibility apps declare which ones should be retrieved and installed into the pipeline of the app, similar to emerging app-in-app ecosystems [50, 103]. In any case, the host app declares the modules in its manifest, where it also states their required privileges and the flow policy, which can hence be inspected and approved (e.g., by the user during app installation). For our prototypical implementation, we create the module sandboxes as dedicated apps as a fixed part of our modified Android image in order to test functional correctness and evaluate our solution in terms of performance overhead (see next Section 5.7).

5.7 Evaluation

In this section, we take two open-source accessibility apps, *TalkBack* [104] and *EVA Facial Mouse* [105], as examples to test the performance of our solution and show how to enhance the privacy protection in accessibility services.

5.7.1 Case Study: TalkBack

We use Google's official screen reader app for visually impaired users, *TalkBack*, to evaluate the protection of on-screen text against leakage. This app has been installed more than 5 billion times according to Google Play Store (see Table A.1 in Appendix A.2). *TalkBack* is a complex app containing multiple modules and multiple preference settings. We focus on its core module—touch-based screen reader with default settings. The accessibility pipeline for this module can be seen in Figure 5.5: the app has an accessibility module and a text-to-speech backend module. Once the user touches the screen, accessibility module collects the textual information about the touched node from the accessibility framework. That information is passed to the text-to-speech component that reads the text aloud via Android's TTS service.

Migration

We build the accessibility module by moving the touch detection logic, which includes accessibility event processing and cursor controls, to an accessibility module in the pipeline. When a touch event is detected, the module outputs the textual information about the UI element at the touch coordinates. Similarly, we establish the backend module here by moving *TalkBack*'s original text-to-speech code to a backend module and exposing the necessary interfaces like `isSpeaking()`, `speak(String)` and `shutdown()` to the host app. To orchestrate this pipeline from the host app, we replace the original local calls in the host app with calls to the API exposed by the two modules (i.e., callbacks for text output from the accessibility module and calls to, e.g., `speak()`). Thus, the host app can forward the text from the touch detection to the text-to-speech logic, each executing in their own sandbox. We made 3k+ LOC changes on a code base 27k+ LOC for this migration.

Privacy enhancement In our design, all modules and host app are running in their own sandbox with distinct permission sets. The accessibility module has the privilege to receive touch events but nothing else, thus, it is unable to scavenge through another app's screen content and leak it. The backend text-to-speech module can only access the TTS API of Android to play the result of the text processing, but cannot leak the text to another sink (e.g., network socket or filesystem). Neither module has the privilege to issue node actions, e.g., pressing buttons in an unauthorized way. By only releasing handles for the output of the accessibility module to the host app, the host app cannot inspect the textual content, which might be privacy-sensitive. Using flow control on those handles, we ensure that the backend module only receives data as input that was generated by the accessibility module.

5.7.2 Case Study: EVA Facial Mouse

We use *EVA Facial Mouse* app to confirm the feasibility of restricting the misuse of node actions. The app provides a virtual mouse that is controlled by facial movements, e.g., if the user cannot use their hands. The accessibility pipeline in Figure 5.6 contains a frontend that uses the camera to capture user intentions and an accessibility module to perform user-intended clicks. The frontend module has access to the device camera and when it detects a head gesture that indicates a click, the coordinates of the virtual mouse on screen will be output. Based on the coordinates, the accessibility module can retrieve the target node from the accessibility framework and perform the actual click on this node.

Migration We put the app's original camera-tracing code to the frontend module and expose necessary callback interfaces, like `onMouseEvent(location, click)`, to the host app. We also allow the necessary accessibility features for node detection based on coordinates and performing click actions on nodes to the accessibility module. As for *TalkBack*, we replace the original calls to the camera and accessibility features inside the host app with calls and callbacks to/from the two modules, such that the host app orchestrates the pipeline and forwards data between the modules. The frontend module traces the user's head movements and outputs the corresponding mouse tracing events, i.e., coordinates of the mouse cursor. To maintain the look and feel of a mouse cursor the host app can in this case dereference the handle to the coordinates data to draw a mouse cursor on screen and also easily allow the user to customize the cursor (e.g., size, color). When a click event is detected by the frontend, the host app invokes the accessibility module with the coordinates for which to retrieve the UI element and to which to issue a click. We changed 1k+ LOC on a code base 9k+ LOC for this app.

Privacy enhancement Again, the modules and host app are in separate sandboxes with distinct permission sets. The frontend module has access to the camera, but nothing else. The accessibility module can retrieve nodes from the accessibility framework based on screen coordinates and issue click actions to those nodes. Applying the text filter to the node infos released to the accessibility module, we prevent that this module learns the content of the UI element (e.g., button label or content of a text view). Further,

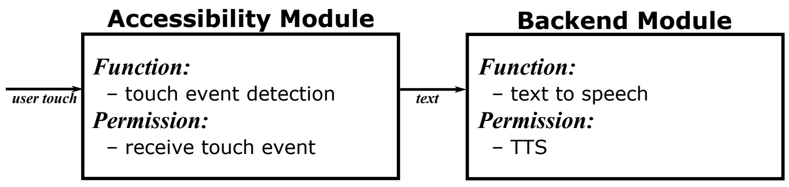


Figure 5.5: Accessibility pipeline for Screen Reader

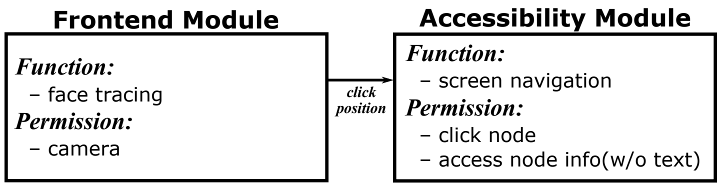


Figure 5.6: Accessibility pipeline for Facial Mouse

neither module can investigate the screen content and hence produce targeted clicks, e.g., to navigate the settings app without user approval to grant permissions or install new apps silently. However, the modules could issue node actions "blindly" and without feedback, e.g., the coordinates are hard-coded in either module, which could succeed in navigating the device surreptitiously when the coordinates fit to the current screen-size and the device screen was in a well-known state (e.g., home screen). A countermeasure to this would rely on our `ActionFilter`, i.e., assuming that the frontend module is trusted and that the coordinates output by this module can be validated against the coordinates of a node when the accessibility module issues a node action. In that case, forging or manipulating coordinates would not succeed.

Further, it should be noted that this app could misuse the camera permission to spy on user input. The trace of cursor coordinates and click events allows the app to monitor where on screen the user clicked. While our solution prevents the app (concretely, the accessibility module) from misusing the accessibility framework to learn *and* leak the information about clicked UI elements, the app can use side-channels to infer this information independently of the accessibility features. For instance, if the host app has a valid assumption about the screen content (e.g., an onscreen keyboard), coordinate trace together with click events would allow the host app to derive which input the user gave (e.g., mapping coordinates with click to the screen position of keys of the soft-keyboard). However, this is purely an abuse of the camera permission and **not** of the accessibility framework. Although being outside of our threat model, our solution could offer a potential solution in this concrete case as well by moving the cursor rendering to a module that cannot leak the derived information and only releasing non-dereferencable handles to the host app.

5.7.3 Performance Overhead

Our framework is deployed on Android version 8.1 on a Pixel 2 XL device. We use the two migrated apps to estimate the performance impact of our framework. We utilize

Table 5.5: Performance evaluation results

Application	Original (ms)	Migrated (ms)	Δ
TalkBack	10.75±1.35	18.55±2.26	7.80 (73%)
EVA Facial Mouse	15.60±1.26	29.50±3.55	13.90 (89%)

Intervals for 95% confidence

microbenchmarking to measure the overhead. Since the runtime of an accessibility operation is affected by complex user interfaces (e.g., time to find a specific node), we develop a dedicated test app with only one `TextView` and one `Button`. Thus, our measurements approximate the upper bound for the overhead, since we minimize the runtime for common operations and thus give more weight to the overhead. We run the test 20 times for the original and migrated versions of the *TalkBack* and *EVA Facial Mouse* app. Table 5.5 summarizes the results.

TalkBack Result We measure the time the screen reader module needs to read the `TextView` text aloud after a user touched on it. We start the measurement as soon as a touch event is detected. The measurement completes when the text-to-speech's `speak` instruction is executed. The average overhead for the migrated app is 7.80ms or about 73%.

EVA Facial Mouse Result We measure the time from click generation in the frontend module until the `onClick()` callback of the target button is triggered. The result shows that the induced overhead is 13.90ms or about 89%.

Summary Although the relative overhead is high, we want to **1)** note again that this is an upper bound since our test app optimizes the common operations and weights the overhead higher and **2)** point out that those affected operations occur in many cases with low frequency and the absolute overhead in our measurements is well below the average human perceptible latency. Thus, while overhead due to the additional IPC between modules and host app was expected, we think the overall overhead is still in an acceptable range.

5.8 Discussion

5.8.1 Limits and challenges

Sandboxing the modules in the pipelines and controlling to which APIs (sources and sinks) they have access together with the opaque, tainted handles for data exchanged between modules provides control over data flows in the same fashion as in similar solution in IoT settings [96]. Thus, we are facing some similar challenges as well as new challenges in protecting the users' privacy.

Indistinguishable data flows Like other solutions, we treat the modules as blackboxes and control the data flows to and from modules. But we cannot control how the modules generate their outputs, and we have only limited means to control the exchanged data (e.g., text filters). As a result, if the data flows including sources and sinks are indistinguishable between a11y and malicious apps, our solution can likely not prevent unauthorized leakage—although, we did not find an example in our study of malicious accessibility apps where this was case, as shown in Section 5.4.

Authorized node actions Further, we face the additional requirement that not only the unwanted leakage of data should be prevented but also unauthorized node actions. The challenge is to connect a node action with a user action. Our current solution tries to validate the parameters of actions (i.e., action filter) but at least limits the effectiveness of malicious node actions by limiting the data on which actions are based (e.g., preventing the reconnaissance of the screen content, see Section 5.7.2).

Off-device processing Accessibility apps can depend on off-device services, for instance, for image or audio processing. As for other information flow control solutions, like [96, 106, 107], the device boundary is a hard boundary for our enforcement. However, by strengthening the sandbox (see below) our solution can provide control over the network destinations (e.g., URL) to which modules can connect and, hence, ensure that only trusted, user-approved services are used as part of the pipeline.

Side-channels We cannot exclude side-channels that can be used by modules to secretly exchange data or that modules use to conduct reconnaissance (e.g. [108]).

Summary While the ideal result would be to prevent all potential leakage of private data and all malicious node actions as described in Section 5.4.4 while at the same time upholding all benign, legitimate a11y app functionality, there currently exist potential cases in which malicious and a11y apps are not distinguishable for our policies. However, compared to stock Android's all-or-nothing protection, our solution provides a trade-off where required assistive apps can function while the potential for misuse of accessibility features is drastically reduced. For instance, a user that requires a facial mouse and allows the corresponding policy might still fall victim to *"blindly"* injected click events but none of the other malware could operate as usual, such as e-banking fraud and content eavesdropping.

5.8.2 Strengthening the sandbox and IFC

An obvious improvement to our solution would be better information flow control along the *entire* data flow even *within* sandboxes. This could help to validate that node actions indeed depend on input generated by user actions or that leaked data does not depend on private data. On Android, taint tracking [106, 107, 109] techniques have been proposed to this end. Unfortunately, taint tracking suffers from a hard to defend reference monitoring. A malicious app can always win by dropping the taint (e.g., native code, indirect control flows) and currently these solutions would only apply to curious-but-honest modules. Further, we currently consider every module to be used only in one pipeline, which makes the information flow control between modules (e.g., the tainting of the handles) and the non-interference within a single module trivial. A more advanced scenario could allow the re-use of modules in different, simultaneously executing pipelines (e.g., re-using pre-installed modules) and in that case an integration of distributed information flow control (e.g., [110] or [96]) would be necessary to ensure non-interference. Our sandboxes rely on the stock Android mechanisms (i.e., *uids* with permissions). However, those only provide a very coarse-grained access control to the application framework or filesystem. Since the way how we start modules from the `PASManagerService` resembles the procedure how virtualized apps are started [50], we could integrate *"module virtualization"* in the future where the `PASManagerService` takes the role of the broker and puts modules into an isolated process (the least privileged execution environment that stock Android supports). Similarly, frameworks [111] for more fine-grained and context-sensitive access control policies could be integrated to provide better control over the functionality and data each module can access from the Android API.

5.8.3 User approval

In our current prototype, we assume that the user approves the policies via the `PASServer` app or writes custom policies that satisfy their individual privacy preferences via this app. We are aware that the history of permission prompts on Android has shown that users are not capable of this and that there is ongoing research into improving the user experience (e.g., seminal work by Porter Felt et al. [112]). Since the user in our solution even has to approve *flows* and the goal of this work is to show that `a11y` and privacy protection do not have to be mutually exclusive *per se*, we defer the question of how to improve the user experience in approving or configuring policies in our solution to future work.

5.8.4 Threats to Validity

We specifically looked for collections of malware samples in actively maintained, popular GitHub projects. However, we cannot guarantee that those collections are the most representative ones for malicious *accessibility* apps. Further, we searched for supposedly benign `a11y` and utility apps by keyword search among the top apps on Google Play Store. Thus, we think our collection of utility apps is representative. Unfortunately, the number of `a11y` apps is limited and many of the top apps are written by Google. Thus, there might be a bias in our collection of `a11y` apps towards Google's software engineering practices.

5.8.5 Utility apps

Our solution reduces the chances for misuse of the accessibility API while preserving the functionality of `a11y` apps. However, utility apps might depend on pipelines that differ from those of `a11y` apps when providing innovative usages of the accessibility framework. For example, password managers take advantage of it to fill-in passwords.

Since this is an abuse of the accessibility features, Google introduced the auto-fill framework [113] as an alternative to support password managers. However, for cases in which no alternative framework or API exists in Android, it remains an open question how to support those utility apps while maintaining a high level of privacy protection or whether those use-cases can be generally implemented in accessibility pipelines as well. For instance, the pipeline for a utility app that automates tedious user actions through sequences of automated button clicks might not be distinguishable enough from malware secretly navigating user interfaces.

5.8.6 Other attacks and privacy issues

Apart from the attacks we analyzed in Section 5.4, other attacks might leverage the accessibility framework as a building block or stepping stone. For instance, the accessibility API might be used for reconnaissance. A typical example is a phishing attack, in which a malicious app uses the accessibility framework to monitor the name of the foreground activity and time the launch a phishing activity. However, in such cases, the accessibility framework is often just the path of least resistance to gather information and alternatives exist (e.g., foreground activities can also be identified via side-channels [108]), thus we did not separately study and evaluate those attacks in our work. Further, our defense relies on proper policies, thus, if the user is involved in setting and granting them, we exclude attacks against the user from our threat model, such as deceptive overlays [90, 93]. Lastly, there exist apps to support impaired or disabled users via crowdsourcing instead of relying on the accessibility framework. For example, camera-based assistive apps to support visually impaired users. Those apps outsource the users' questions, e.g., about their physical surroundings, to volunteers with whom the users have to share sensitive information, such as photo or video stream. Prior work [114] investigated the privacy concerns raised in using camera-based assistive apps under different scenarios. Their results confirm the request by dependent users for privacy protection in using assistive technologies, which we take as further motivation for our research although those particular cases of sharing camera data with volunteers are not covered by our work. Similarly, other data stealing attacks, such as taking screenshots or recording audio that do not rely on the accessibility framework are out of scope of what we can defend against.

5.9 Related Work

We briefly present the related works to our work on enhancing the privacy of Android's accessibility framework.

Process-based privilege separation on Android A few solutions separate sensitive or untrusted components into isolated processes to mitigate privacy violations. Works focusing on advertisement libraries [27, 26, P1] demonstrated different solutions to isolate said libraries from their host apps and privilege separate them. Roesner et al. [115] sandboxed untrusted UI components in isolated processes to support secure UI embedding. Davidson et al. [33] provided a dedicated `WebView` service app to protect host apps from untrusted web content. Starting with Android O, Google officially put the `WebView` renderer into an isolated process [116]. Other works privilege separate entire apps, e.g., Backes et al. [50] create a virtualized environment for untrusted apps and, similarly, Bianchi et al. [51] demonstrated an approach that sandboxes an untrusted app inside a separate non-privileged context to enforce privacy and security policies. Our work transfer those concepts to the accessibility framework by sandboxing the code modules that form an accessibility pipeline.

Information flow control in IoT applications Closest to our work is FlowFence [96], which introduced information flow control for IoT apps to prevent unwanted data leakage. It introduced the concepts of *quarantined* modules and *opaque handles* that we also used in our implementation. In contrast to FlowFence, our flows are very simple and linear. For instance, in FlowFence multiple flows might converge on the same module, necessesitating taint sets for modules, and modules can set custom taints on output to prevent their data from reaching certains sinks. Our modules only consume output from a single predecessor module within a short pipeline for which the user sets the policy. Thus, while we could support the same taint arithmetic and taint sets as FlowFence, this is currently not necessary and simplifies our setup, avoiding the issue of overtainting module sandboxes. On the other hand, our solution has to additionally deal with the problem of authorizing actions by modules. We addressed this by adopting the concept of *recognizers* [95] and using data flow control to limit the information needed for (effective) malicious actions.

5.10 Conclusion

Android's accessibility framework is a powerful service intended to allow assistive apps in supporting impaired and disabled users in navigating their devices. Unfortunately, the service is also a popular building block for utility and malevolent apps that do not apply accessibility features as originally intended and might violate the users' privacy. Existing defenses in stock Android force users and app developers to choose between inclusiveness and privacy protection. To improve on this situation, we propose a privacy-enhanced accessibility framework forward. By representing a11y logic as pipelines, sandboxing every code module in a pipeline, and enforcing flow constraints, our solution allows more fine-grained control over accessibility features and reduces the attack surface while upholding the functionality of a11y apps. We showcase the feasibility of our solution by migrating two a11y apps. We also discuss the shortcomings of our approach and hope this work will raise further interest in building solutions that protect a particular dependent user group.

6 Conclusion

Untrusted application components, including third-party libraries and app components that abuse the privacy-sensitive framework APIs, pose many privacy and security problems to application users. The permission model and the sandboxing mechanism under the current Android security model are ineffective in protecting user data from being violated by untrusted app components. In addition, the lack of timely update support for flawed third-party components in the current Android app ecosystem exposes a large vulnerability window to the public, which further worsens the risk of privacy leakage.

This dissertation presents a line of works that mitigate the privacy and security threats originated from untrusted application components on Android. The emphasis of this dissertation lies on either optimizing the integration model for untrusted app components to provide more fine-grained privilege restriction straightforwardly or otherwise providing timely remediations to those components once they are detected to be vulnerable or exploitable. Towards implementing a more fine-grained privilege access control mechanism, we borrow the idea of privilege separation from existing works where the privilege sets of the untrusted components and other parts of the app are distinct and capable of being policed separately. This dissertation chooses to split the code of the untrusted components from other parts of the app code and make them run in different processes where each process has its own *uid* and permission set. With the Android sandboxing mechanism, strong trustworthy boundaries can be established among the untrusted processes and other app processes, and then sensitive data access in different sandboxes can now be managed separately. The primary challenge of this line of works is to rebuild the communication channel between those isolated sandboxes to synchronize the now physically separated untrusted components and the other parts of the app across processes while maintaining the original app's functionality and user interface. Aside from that, another critical aspect of these works is to apply a least-privileged privacy policy to each sandbox, which allows for fine-grained control of privileges to different compartments, thus significantly reducing the sensitive data exposure to untrusted components. Apart from implementing more fine-grained privilege access control mechanisms, this dissertation also focuses on the remediation measurement for known vulnerable third-party components, which can also help mitigate privacy and security threats originating from untrusted app components. More specifically, this work explores the possibility of reducing the open vulnerability window caused by the untrusted components through an improved app update ecosystem.

Our work on mitigating security and privacy threats from untrusted application components consists of three peer-reviewed publications [P1, P2, P3]. Among them, `CompARTist` (see Chapter 3) is a compiler-based library compartmentalization solution that utilizes the sandboxing-based privilege separation technique to prevent host data from being compromised by third-party advertisement libraries. Building on top of Android's *dex2oat* on-device compiler, `CompARTist` transforms the target application and replaces the original local library calls with inter-process communication calls to the remote library service app seamlessly during a recompilation. Through this newly established IPC channel, the remote advertisement service app can sync with the host app, and thus the advertising functionality and user interface of the original application can be preserved while effectively mitigating privacy and security threats from the advertising library without modifying any systems or applications. While `CompARTist` protects user privacy from overly curious or malicious third-party libraries by altering the integration model for those libraries, our second work, `Up2Crash` (see Chapter 4), emphasizes minimizing those threats from vulnerable third-party libraries within the original in-app library integration model. More specifically, `Up2Crash` focuses on reducing the open vulnerability window caused by vulnerable third-party libraries by breaking through the bottleneck of the library update chain inside the Android app ecosystem and updating vulnerable libraries to their successor (patched) versions timely without involving app developers. It explores the feasibility of an API-compatibility-based library update solution that updates the outdated libraries through drop-in replacement as proposed by prior work [2]. Multiple dynamic tests on real-world applications uncovered intricate factors that impede a drop-in replacement of libraries. Further case studies verified the prevalence of those discovered issues in other libraries and pinpointed entangled library dependencies as the main challenging issue of the drop-in replacement solution. `Up2Crash` is the first work to study the factors influencing an API-compatibility-based library update solution. Our findings provide valuable insights for future improvements to the library update ecosystem or library update tool development. Both `CompARTist` and `Up2Crash` aim to mitigate privacy and security threats from libraries created by untrusted third-party. However, there also exists privacy leakage from untrusted components implemented by the app developer themselves. The third work of this dissertation is dedicated to reducing the risk of privacy leaks due to the misuse of powerful accessibility features by untrusted application components through a privacy-enhanced accessibility framework (see Chapter 5). In this work, the complete accessibility logic is treated as a pipeline consisting of multiple modules. Similar to `CompARTist`, the privilege separation technique is also applied here, enabling a more fine-grained privilege control over the usage of accessibility features by loading all code modules inside the pipeline to different sandboxes. Moreover, we deployed a flow control mechanism on this pipeline to further improve the management of sensitive data transfers between modules. This approach upholds the functionality of accessibility apps while significantly reducing the attack surface. Real-world app migration cases also proved the feasibility of our solution.

Future Research Directions We note several future research directions for this dissertation. All solutions in this line of works mitigate the privacy and security threats through on-device measures. These measures prohibit unwanted app components from accessing sensitive user data, but they do not prevent data from being leaked by apps that involve off-device processing, for example, image recognition, as discussed in Section 5.8.1. Restricting the remote data destinations with a whitelist could be a potential solution, but again this involves the methodology used to generate this whitelist. Ranking-based whitelist makes the solution inflexible and non-adaptable, e.g., some post-ranking changes in the remote server may cause information leakage, while pure user-decision-based solutions are not reliable enough. Based on this situation, our vision is to extend the protection domain of the security mechanism beyond the device, for example, to a hardware-secured cloud platform. Under this new architecture, remote services can preserve their usability by submitting data preprocessing modules to this platform and let these modules work as the gateway between user data and remote service. The secure hardware will protect the intellectual property of the data preprocessing modules. The data preprocessing module can minimize and desensitize the raw data from users, thus enhancing user privacy in using those off-device services.

Apart from that, we also notice some application scenarios of our solution beyond mobile apps, e.g., IoT devices. Chapter 4 highlights the bottleneck of library update in the current Android software ecosystem. When it comes to IoT devices, this problem has not diminished but has become more complex. IoT devices usually rely on companion apps to prompt update installation requests to users and further install the updated firmware. Considering the data sensitivity in IoT devices, such as fitness monitors, and the tedious one-by-one upgrading and permission management work, we do see the need to have an easy-to-use mechanism, e.g., a centralized auto-updating and permission management system service, on the current Android to keep the firmware of these IoT devices updated in time.

Bibliography

Author's Papers for this Thesis

[P1] Huang, J., Schranz, O., Bugiel, S., and Backes, M. The art of app compartmentalization: compiler-based library privilege separation on stock android. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017, 1037–1049.

[P2] Huang, J., Borges, N., Bugiel, S., and Backes, M. Up-to-crash: evaluating third-party library updatability on android. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, 15–30.

[P3] Huang, J., Backes, M., and Bugiel, S. A11y and privacy don't have to be mutually exclusive: constraining accessibility service misuse on android. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

Other references

[1] Backes, M., Bugiel, S., Schranz, O., Styp-Rekowsky, P. von, and Weisgerber, S. Artist: the android runtime instrumentation and security toolkit. In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, 481–495.

[2] 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 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, 2187–2200.

[3] Borges, N. P., Hotzkow, J., and Zeller, A. Droidmate-2: a platform for android test generation. In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2018, 916–919.

[4] *Number of smartphone users worldwide from 2016 to 2023*. `https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide`. Accessed: 2021-04-23.

[5] *Number of available applications in the Google Play Store from December 2009 to December 2020*. `https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/`. Accessed: 2021-04-23.

[6] *Cloud-based protections*. `https://developers.google.com/android/play-protect/cloud-based-protections`. Accessed: 2021-09-30.

[7] *LibScout*. https://github.com/reddr/LibScout. Accessed: 2021-04-23.

[8] *Open Handset Alliance Releases Android SDK*. http://www.openhandsetalliance.com/press_111207.html. Accessed: 2021-05-31.

[9] *HTC Dream*. https://en.wikipedia.org/wiki/HTC_Dream. Accessed: 2021-05-31.

[10] *Mobile Operating System Market Share Worldwide*. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 2021-04-23.

[11] *OpenBinder*. http://www.angryredplanet.com/~hackbod/openbinder/docs/html/. Accessed: 2021-09-31.

[12] *Build more accessible apps*. https://developer.android.com/guide/topics/ui/accessibility. Accessed: 2021-04-23.

[13] *AccessibilityService*. https://developer.android.com/reference/android/accessibilityservice/AccessibilityService. Accessed: 2021-04-23.

[14] *AccessibilityServiceInfo*. https://developer.android.com/reference/android/accessibilityservice/AccessibilityServiceInfo. Accessed: 2021-04-23.

[15] *android:importantForAccessibility*. https://developer.android.com/reference/android/view/View.html#attr_android:importantForAccessibility. Accessed: 2021-04-23.

[16] Thomas, D. R., Beresford, A. R., and Rice, A. Security metrics for the android ecosystem. In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. 2015, 87–98.

[17] *What happened to the Android Update Alliance?* https://arstechnica.com/gadgets/2012/06/what-happened-to-the-android-update-alliance/. Accessed: 2021-04-23.

[18] *Update your Android apps*. https://support.google.com/googleplay/answer/113412?hl=en. Accessed: 2021-04-23.

[19] *Improve your code with lint checks*. https://developer.android.com/studio/write/lint. Accessed: 2021-04-23.

[20] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating user privacy in android ad libraries. In: *Workshop on Mobile Security Technologies (MoST)*. Vol. 10. Citeseer. 2012.

[21] Demetriou, S., Merrill, W., Yang, W., Zhang, A., and Gunter, C. A. Free for all! assessing user data exposure to advertising libraries on android. In: *NDSS*. 2016.

[22] Son, S., Kim, D., and Shmatikov, V. What mobile ads know about mobile users. In: *NDSS*. Citeseer. 2016.

[23] Meng, W., Ding, R., Chung, S. P., Han, S., and Lee, W. The price of free: privacy leakage in personalized mobile in-apps ads. In: *NDSS*. 2016.

[24] 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 communications security.* 2014, 483–494.

[25] Backes, M., Bugiel, S., Styp-Rekowsky, P. von, and Wißfeld, M. Seamless in-app ad blocking on stock android. In: *2017 IEEE Security and Privacy Workshops (SPW).* IEEE. 2017, 163–168.

[26] Pearce, P., Felt, A. P., 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 Communications Security.* 2012, 71–72.

[27] Shekhar, S., Dietz, M., and Wallach, D. S. Adsplit: separating smartphone advertising from applications. In: *21st USENIX Security Symposium (USENIX Security 12).* 2012, 553–567.

[28] Zhang, X., Ahlawat, A., and Du, W. Aframe: isolating advertisements from mobile applications in android. In: *Proceedings of the 29th Annual Computer Security Applications Conference.* 2013, 9–18.

[29] Seo, J., Kim, D., Cho, D., Shin, I., and Kim, T. Flexdroid: enforcing in-app privilege separation in android. In: *NDSS.* 2016.

[30] Liu, B., Liu, B., Jin, H., and Govindan, R. Efficient privilege de-escalation for ad libraries in mobile apps. In: *Proceedings of the 13th annual international conference on mobile systems, applications, and services.* 2015, 89–103.

[31] Zhou, Y., Patel, K., Wu, L., Wang, Z., and Jiang, X. Hybrid user-level sandboxing of third-party android apps. In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* 2015, 19–30.

[32] Sun, M. and Tan, G. Nativeguard: protecting android applications from third-party native libraries. In: *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks.* 2014, 165–176.

[33] Davidson, D., Chen, Y., George, F., Lu, L., and Jha, S. Secure integration of web content and applications on commodity mobile operating systems. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* 2017, 652–665.

[34] *Ad-blocking for your Android.* `https://adaway.org/`. Accessed: 2021-04-23.

[35] *Surf the Web Ad-Free and Safely. Shield up!* `https://adguard.com/en/welcome.html`. Accessed: 2021-04-23.

[36] *Surf the web with no annoying ads.* `https://adblockplus.org/`. Accessed: 2021-04-23.

[37] *Browse fast, safe and free of annoying ads with Adblock Browser.* `https://adblockbrowser.org/`. Accessed: 2021-04-23.

[38] Erlingsson, U. *The inlined reference monitor approach to security policy enforcement.* Tech. rep. Cornell University, 2003.

[39] 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 permissions in android applications. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices.* 2012, 3–14.

[40] Backes, M., Gerling, S., Hammer, C., Maffei, M., and Styp-Rekowsky, P. von. Appguard–enforcing user requirements on android apps. In: *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems.* Springer. 2013, 543–548.

[41] Davis, B. and Chen, H. Retroskeleton: retrofitting android apps. In: *Proceeding of the 11th annual international conference on Mobile systems, applications, and services.* 2013, 181–192.

[42] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-arm-droid: a rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies* 2012, 2 (2012), 1–7.

[43] Rasthofer, S., Arzt, S., Lovat, E., and Bodden, E. Droidforce: enforcing complex, data-centric, system-wide policies in android. In: *2014 Ninth International Conference on Availability, Reliability and Security.* IEEE. 2014, 40–49.

[44] Xu, R., Saïdi, H., and Anderson, R. Aurasium: practical policy enforcement for android applications. In: *21st USENIX Security Symposium (USENIX Security 12).* 2012, 539–552.

[45] *javap - The Java Class File Disassembler.* `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html`. Accessed: 2021-09-01.

[46] Schreiber, T. Android binder. *A shorter, more general work, but good for an overview of Binder. http://www. nds. rub. de/media/attachments/files/2012/03/binder. pdf* (2011).

[47] *UI/Application Exerciser Monkey.* `https://developer.android.com/studio/test/monkey.html`. Accessed: 2021-04-23.

[48] *Set up Google Play services.* `https://developers.google.com/android/guides/setup`. Accessed: 2021-04-23.

[49] *interstitial.* `https://developers.google.com/admob/android/interstitial#some_best_practices`. Accessed: 2021-07-05.

[50] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and Styp-Rekowsky, P. von. Boxify: full-fledged app sandboxing for stock android. In: *24th USENIX Security Symposium (USENIX Security 15).* 2015, 691–706.

[51] Bianchi, A., Fratantonio, Y., Kruegel, C., and Vigna, G. Njas: sandboxing unmodified applications in non-rooted devices running stock android. In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices.* 2015, 27–38.

[52] *Guava: Google Core Libraries for Java.* `https://github.com/google/guava`. Accessed: 2021-04-23.

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

[54] Bhoraskar, R., Han, S., Jeon, J., Azim, T., Chen, S., Jung, J., Nath, S., Wang, R., and Wetherall, D. Brahmastra: driving apps to test the security of third-party components. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, 1021–1036.

[55] Jamrozik, K. and Zeller, A. Droidmate: a robust and extensible test generator for android. In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. 2016, 293–294.

[56] Wei, T., Zhang, Y., Xue, H., Zheng, M., Ren, C., and Song, D. Sidewinder: targeted attack against android in the golden age of ad libraries. *Black Hat* 14 (2014).

[57] *AppLovin Security Notice*. `https://blog.applovin.com/applovin-security-notice/`. Accessed: 2021-04-23.

[58] *JS-Binding-Over-HTTP Vulnerability and JavaScript Sidedoor: Security Risks Affecting Billions of Android App Downloads*. `https://www.fireeye.com/blog/threat-research/2014/01/js-binding-over-http-vulnerability-and-javascript-sidedoor.html`. Accessed: 2021-04-23.

[59] Watanabe, T., Akiyama, M., Kanei, F., Shioji, E., Takata, Y., Sun, B., Ishi, Y., Shibahara, T., Yagi, T., and Mori, T. Understanding the origins of mobile app vulnerabilities: a large-scale measurement study of free and paid apps. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, 14–24.

[60] Salza, P., Palomba, F., Di Nucci, D., D'Uva, C., De Lucia, A., and Ferrucci, F. Do developers update third-party libraries in mobile apps? In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, 255–265.

[61] Thomas, D. R., Beresford, A. R., Coudray, T., Sutcliffe, T., and Taylor, A. The lifetime of android api vulnerabilities: case study on the javascript-to-java interface. In: *Cambridge International Workshop on Security Protocols*. Springer. 2015, 126–138.

[62] Beresford, A. R. Whack-a-mole security: incentivising the production, delivery and installation of security updates. In: *IMPS@ ESSoS*. 2016, 9–10.

[63] Zhang, M. and Yin, H. Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In: *NDSS*. Citeseer. 2014.

[64] Mulliner, C., Oberheide, J., Robertson, W., and Kirda, E. Patchdroid: scalable third-party security patches for android devices. In: *Proceedings of the 29th Annual Computer Security Applications Conference*. 2013, 259–268.

[65] Chen, Y., Li, Y., Lu, L., Lin, Y.-H., Vijayakumar, H., Wang, Z., and Ou, X. Instaguard: instantly deployable hot-patches for vulnerable system programs on android. In: *2018 Network and Distributed System Security Symposium (NDSS'18)*. 2018.

[66] You, W., Liang, B., Shi, W., Zhu, S., Wang, P., Xie, S., and Zhang, X. Reference hijacking: patching, protecting and analyzing on unmodified and non-rooted android devices. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE. 2016, 959–970.

[67] Zhang, X., Zhang, Y., Li, J., Hu, Y., Li, H., and Gu, D. Embroidery: patching vulnerable binary code of fragmentized android devices. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, 47–57.

[68] Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., and Wei, T. Adaptive android kernel live patching. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, 1253–1270.

[69] Duan, R., Bijlani, A., Ji, Y., Alrawi, O., Xiong, Y., Ike, M., Saltaformaggio, B., and Lee, W. Automating patching of vulnerable open-source software versions in application binaries. In: *NDSS*. 2019.

[70] Kim, D., Nam, J., Song, J., and Kim, S. Automatic patch generation learned from human-written patches. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, 802–811.

[71] Long, F. and Rinard, M. Automatic patch generation by learning correct code. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, 298–312.

[72] Li, B., Zhang, Y., Li, J., Feng, R., and Gu, D. Appcommune: automated third-party libraries de-duplicating and updating for android apps. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, 344–354.

[73] *Amigo*. `https://github.com/eleme/Amigo`. Accessed: 2021-04-23.

[74] *AndFix*. `https://github.com/alibaba/AndFix`. Accessed: 2021-04-23.

[75] *Tinker*. `https://github.com/Tencent/tinker`. Accessed: 2021-04-23.

[76] *Droid Plugin*. `https://github.com/DroidPluginTeam/DroidPlugin`. Accessed: 2021-04-23.

[77] Choudhary, S. R., Gorla, A., and Orso, A. Automated test input generation for android: are we there yet?(e). In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, 429–440.

[78] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M. Using gui ripping for automated testing of android applications. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2012, 258–261.

[79] Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. 2012, 93–104.

[80] Mao, K., Harman, M., and Jia, Y. Sapienz: multi-objective automated testing for android applications. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, 94–105.

[81] Wong, M. Y. and Lie, D. Intellidroid: a targeted input generator for the dynamic analysis of android malware. In: *NDSS*. Vol. 16. 2016, 21–24.

[82] *Build and run your app*. `https://developer.android.com/studio/run#instant-run`. Accessed: 2021-04-23.

[83] *Gradle Build Tool*. `https://gradle.org/`. Accessed: 2021-09-31.

[84] *Android Ad Network statistics and market share*. `https://www.appbrain.com/stats/libraries/ad`. Accessed: 2021-04-23.

[85] Diao, W., Zhang, Y., Zhang, L., Li, Z., Xu, F., Pan, X., Liu, X., Weng, J., Zhang, K., and Wang, X. Kindness is a risky business: on the usage of the accessibility apis in android. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 2019, 261–275.

[86] *Android Trojan steals money from PayPal accounts even with 2FA on*. `https://www.welivesecurity.com/2018/12/11/android-trojan-steals-money-paypal-accounts-2fa/`. Accessed: 2021-04-23.

[87] *Skygofree — a Hollywood-style mobile spy*. `https://www.kaspersky.com/blog/skygofree-smart-trojan/20717`. Accessed: 2021-04-23.

[88] *Anubis Strikes Again: Mobile Malware Continues to Plague Users in Official App Stores*. `https://securityintelligence.com/anubis-strikes-again-mobile-malware-continues-to-plague-users-in-official-app-stores/`. Accessed: 2021-04-23.

[89] Jang, Y., Song, C., Chung, S. P., Wang, T., and Lee, W. A11y attacks: exploiting accessibility in operating systems. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, 103–115.

[90] Fratantonio, Y., Qian, C., Chung, S. P., and Lee, W. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, 1041–1057.

[91] Kraunelis, J., Chen, Y., Ling, Z., Fu, X., and Zhao, W. On malware leveraging the android accessibility framework. In: *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer. 2013, 512–523.

[92] Kalysch, A., Bove, D., and Müller, T. How android's ui security is undermined by accessibility. In: *Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium*. 2018, 1–10.

[93] Yan, Y., Li, Z., Chen, Q. A., Wilson, C., Xu, T., Zhai, E., Li, Y., and Liu, Y. Understanding and detecting overlay-based android malware at market scales. In: *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 2019, 168–179.

[94] Naseri, M., Borges Jr, N. P., Zeller, A., and Rouvoy, R. Accessileaks: investigating privacy leaks exposed by the android accessibility service (2019).

[95] Jana, S., Molnar, D., Moshchuk, A., Dunn, A., Livshits, B., Wang, H. J., and Ofek, E. Enabling fine-grained permissions for augmented reality applications with recognizers. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, 415–430.

[96] Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. Flowfence: practical data protection for emerging iot application frameworks. In: *25th USENIX security symposium (USENIX Security 16)*. 2016, 531–548.

[97] *Android-Malwares*. `https://github.com/hxp2k6/Android-Malwares`. Accessed: 2021-04-23.

[98] *Android Malware Samples*. `https://github.com/ashishb/android-malware`. Accessed: 2021-04-23.

[99] *Android Malware - 2018*. `https://github.com/sk3ptre/AndroidMalware_2018`. Accessed: 2021-04-23.

[100] *AndroidMalware_2019*. `https://github.com/sk3ptre/AndroidMalware_2019`. Accessed: 2021-04-23.

[101] *Google pauses removal of apps that want to use accessibility services*. `https://www.zdnet.com/article/google-pauses-crackdown-of-accessibility-api-apps/`. Accessed: 2021-04-23.

[102] *VirusTotal*. `https://support.virustotal.com/hc/en-us`. Accessed: 2021-04-23.

[103] Lu, H., Xing, L., Xiao, Y., Zhang, Y., Liao, X., Wang, X., and Wang, X. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, 569–585.

[104] *TalkBack*. `https://github.com/google/talkback`. Accessed: 2021-04-23.

[105] *EVA Facial Mouse*. `https://github.com/cmauri/eva_facial_mouse`. Accessed: 2021-04-23.

[106] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[107] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: *Proceedings of the 18th ACM conference on Computer and communications security*. 2011, 639–652.

[108] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into your app without actually seeing it: ui state inference and novel android attacks. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, 1037–1052.

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

[110] Nadkarni, A., Andow, B., Enck, W., and Jha, S. Practical DIFC enforcement on android. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, 1119–1136.

[111] Heuser, S., Nadkarni, A., Enck, W., and Sadeghi, A.-R. ASM: a programmable interface for extending android security. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, 1005–1019.

[112] Felt, A. P., Egelman, S., Finifter, M., Akhawe, D., Wagner, D. A., et al. How to ask for permission. *HotSec* 12 (2012), 7–7.

[113] *Autofill framework*. `https://developer.android.com/guide/topics/text/autofill`. Accessed: 2021-04-23.

[114] Akter, T., Dosono, B., Ahmed, T., Kapadia, A., and Semaan, B. " i am uncomfortable sharing what i can't see": privacy concerns of the visually impaired with camera based assistive applications. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, 1929–1948.

[115] Roesner, F. and Kohno, T. Securing embedded user interfaces: android and beyond. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, 97–112.

[116] *What's new in WebView security*. `https://android-developers.googleblog.com/2017/06/whats-new-in-webview-security.html`. Accessed: 2021-04-23.

A
Appendix

A.1 APIs of CompARTist's communication channel

```
void invokeListenerCallbackHelper(int objectId, String method);
void invokeListenerCallbackHelper_1(int objectId, String method, in WrapClass 
    param);
void invokeListenerCallbackHelper_2(int objectId, String method, in WrapClass 
    param_1, in WrapClass param_2);
void invokeListenerCallbackHelper_3(int objectId, String method, in WrapClass 
    param_1, in WrapClass param_2, in WrapClass param_3);
void invokeListenerCallbackHelper_4(int objectId, String method, in WrapClass 
    param_1, in WrapClass param_2, in WrapClass param_3, in WrapClass param_4);
```

Listing A.1: Callback API

```
WrapClass getStaticFieldService(String ctype, String field);
WrapClass invokeStaticMethodService_2(String ctype, String method,in WrapClass[] 
    params);
WrapClass invokeStaticMethodService(String ctype, String method);
WrapClass invokeVirtualMethodService_2(String ctype, String method, in WrapClass 
    object, in WrapClass[] params);
WrapClass invokeVirtualMethodService(String ctype, String method, in WrapClass 
    object);
WrapClass newInstanceService_2(String ctype, in WrapClass[] params);
WrapClass newInstanceService(String ctype);
```

Listing A.2: Advertisement Invocation API

```
void removeWindow(int viewId, boolean destroy);
void createWindow(int viewId, in Rect rect);
void updateWindow(int viewId, in Rect rect);
```

Listing A.3: Lifecycle API

A.2 Accessibility App Sample Set

Table A.1: A11y App Sample Set (Google Play Store)

Package	#Installed
com.google.android.marvin.talkback	5,000,000,000+
com.sesame.phone_nougat	10,000+
com.crea_si.eviacam.service	1,000,000+
com.google.audio.hearing.visualization.accessibility.scribe	50,000,000+
com.google.android.apps.accessibility.voiceaccess	1,000,000+

Table A.2: Utility App Sample Set (Google Play Store)

Package	**#Installed**
com.amazon.tahoe	1,000,000+
com.antivirus	100,000,000+
com.antivirus.tablet	10,000,000+
com.apusapps.emo_launcher	100,000+
com.apusapps.launcher	100,000,000+
com.avast.android.cleaner	10,000,000+
com.avast.android.mobilesecurity	100,000,000+
com.avg.cleaner	50,000,000+
com.bitdefender.security	5,000,000+
com.bitspice.automate	500,000+
com.cleanmaster.mguard	1,000,000,000+
com.eset.ems2.gp	10,000,000+
com.eset.parental	100,000+
com.gau.go.launcherex	100,000,000+
com.italia.autovelox.autoveloxfissiemoibli	1,000,000+
com.kaspersky.safekids	500,000+
com.kms.free	50,000,000+
com.ksmobile.launcher	100,000,000+
com.lastpass.lpandroid	5,000,000+
com.lionmobi.battery	50,000,000+
com.mcafee.security.safefamily	100,000+
com.microsoft.launcher	10,000,000+
com.oneapp.max.cleaner.booster	10,000,000+
com.piriform.ccleaner	50,000,000+
com.pleco.chinesesystem	1,000,000+
com.server.auditor.ssh.client	500,000+
com.teslacoilsw.launcher	50,000,000+
com.wsandroid.suite	10,000,000+
dreamy.earth.theme.natural.launcher	1,000,000+
galaxy.iphone.hd.wallpaper.live.screen.lock	10,000,000+
ginlemon.flowerfree	10,000,000+
mobi.infolife.appbackup	10,000,000+
org.malwarebytes.antimalware	10,000,000+
panda.keyboard.emoji.theme	100,000,000+
red.love.rose.theme.valentine.launcher	1,000,000+

Table A.3: Malicious App Sample Set (Github)

MD5	Classification (VirusTotal-Alibaba)
007ae04ac52f17d5d637f2c41658f824	TrojanSpy:Android/Banker.a30eb151
03e5d8ece783245b28cb97373e739842	Trojan:Android/Agent.3fc9b0c7
042f2f3a0df4aef0460d1ee74f1df033	Backdoor:Android/Agent.8f28ba9e
09b60aa78291e7ef8b0ddfc261afb9f9	TrojanDropper:Android/Skeeyah.a026644f
10f8097ef0db6adbed3b314055491ca4	Trojan:Android/Rootnik.efbca116
1512c3fa688ca107784b3c93cd9f3526	TrojanDropper:Android/Hqwar.657ae279
18a3c09ce58b3db05cf248730adb6bd0	TrojanDropper:Android/Hqwar.9e0b0668
2254002370c03cf14c3eabb27b3b826d	TrojanBanker:Android/Anubis.58e63764
2f07c9b2a67104f8bc08d831c8922b6a	TrojanBanker:Android/Riltok.32dfd36e
31ba565fcc1060ad848769e0b5b70444	Trojan:Android/Agent.4c52deda
39fca709b416d8da592de3a3f714dce8	Trojan:Android/Skygofree.355eb294
3b07862da0b78632d8e4486444adbbfd	Backdoor:Android/Agent.3ebcdecc
3ffedf4759a001417084c64db48b549a	TrojanSpy:Android/AndroRAT.afe389c9
4aea3ec301b3c0e6d813795ca7e191bb	TrojanSpy:Android/Donot.60880405
519018ecfc50c0cf6cd0c88cc41b2a69	TrojanSpy:Android/ZooPark.436e912a
51f388f9ca606812d7fb4d5330e42ce7	TrojanBanker:Android/Anubis.75cc2361
55366b684ce62ab7954c74269868cd91	Trojan:Android/Ztorg.6ff5f73b
5cc953f25deeff951c38a5c118a81fe9	Trojan:Android/Agent.008476da
63a56f3867ef4b4a3cf469e81496aee7	TrojanDropper:Android/Agent.ac60e49b
647f6b2503205dd1f1da5ea490b6c71f	Trojan:Android/Rootnik.977d3960
64e374807d87102cfc27489a91f8a13d	AdWare:Android/Batmob.cbf4dda9
6a3ae5a916bc109e0186b40093084a78	Backdoor:Android/Agent.63d66ab9
6c39197bb2c2fd5fc9253ed18467d0d7	Backdoor:Android/Brata.7b8ffc88
70a937b2504b3ad6c623581424c7e53d	Trojan:Android/Skygofree.f9b277e6
71b80c162001f9d2f4872f2efb7431fe	TrojanSpy:Android/AndroRAT.a2734e43
75f1117cabc55999e783a9fd370302f3	TrojanBanker:Android/Banker.4650457b
880540d10cca559f68db96314f206225	Trojan:Android/Rootnik.1fce124c
8a266e277c61ffd6afa3d15b8691b9fb	Backdoor:Android/Agent.48e611ae
8a9540fa5541054074d1efdc7729da43	Backdoor:Android/Lucbot.5aac9302
8d1f5637dc0bc76064d6f3283482a7c5	Trojan:Android/Agent.7c517cfb
8df5b22cabc10423533884da7648e982	TrojanBanker:Android/Asacub.3fc31d6b
91f0daa8cb837a9d3e815da8db999a08	TrojanSpy:Android/Banker.35c71d45
957ce53315496083a43c6765f5ed9e42	TrojanSpy:Android/AndroRAT.d9b0b7c8
9ae53ef2a4f2d40b06cc85e5c3778f48	Trojan:Android/Agent.14c930cd
a287a434a0d40833d3ebf5808950b858	Trojan:Android/Skygofree.639ce6ec
a2a921c0e8a9171300a805c5b1df78b8	TrojanSpy:Android/Banker.f9398502
a384a27681df0ed1732d4346f6c52d0a	TrojanBanker:Android/Generic.ba1d86be
a44a9811db4f7d39cac0765a5e1621ac	Trojan:Android/Agent.34c921b3
a8a8479dab8fbdee1fb058b8de97e89b	Trojan:Android/Agent.a87db02a
a962759a71f899a9bbe4d27790e91b00	Backdoor:Android/Lucbot.69116e3e
a97eb28853eeeecffb749bf49b68af55	TrojanSpy:Android/AndroRAT.6fd591ab
ac613a7dee1ee8c47321403ab8fa1372	Trojan:Android/Agent.f483d3f4
ac67f1b22d6c7812003609de284a9ad9	TrojanDropper:Android/Hqwar.20f8d210
ac92258ff3395137dd590af36ca2d8c9	Trojan:Android/Agent.c1ade2d1
c148c63c974e2312d8f847d07242a86b	TrojanBanker:Android/Anubis.65b2e27e
c580e7807fbd18106d2659af3cc58f8d	AdWare:Android/Gibdy.2f426bf5
cdf10316664d181749a8ba90a3c07454	TrojanSpy:Android/Donot.d27afe4a
d0641eb22198c346af6c22284fca38a6	TrojanBanker:Android/Riltok.b7c88ed6
d3f53bcf02ede4adda304fc7f03a2000	TrojanSpy:Android/Donot.ecf77e96
d6ef4e16701b218f54a2a999af47d1b4	TrojanBanker:Android/Banker.a4cbd698
dc74daf70afc181471f41fd910a0dec0	TrojanDropper:Android/Hqwar.ef5f2c4a
e105b0fd0eadc5db26bf979e4e96007c	Trojan:Android/Rootnik.1c8d7a29
e4187a74e6bef1a8cd30116500ed10f8	TrojanBanker:Android/Banker.3457c734
ef8493089deecbef6e459434ec7fee0b	TrojanDropper:Android/Hqwar.04dcccff
ffacd0a770aa4faa261c903f3d2993a2	TrojanBanker:Android/Banker.522c0eb4