

REAPER: Real-time App Analysis for Augmenting the Android Permission System

Michalis Diamantaris
FORTH, Greece
diamant@ics.forth.gr

Elias P. Papadopoulos
FORTH, Greece
php@ics.forth.gr

Evangelos P. Markatos
FORTH, Greece
markatos@ics.forth.gr

Sotiris Ioannidis
FORTH, Greece
sotiris@ics.forth.gr

Jason Polakis
Univ. of Illinois at Chicago, USA
polakis@uic.edu

ABSTRACT

Android's app ecosystem relies heavily on third-party libraries as they facilitate code development and provide a steady stream of revenue for developers. However, while Android has moved towards a more fine-grained run time permission system, users currently lack the required resources for deciding whether a specific permission request is actually intended for the app itself or is requested by possibly dangerous third-party libraries.

In this paper we present Reaper, a novel dynamic analysis system that traces the permissions requested by apps in real time and distinguishes those requested by the app's core functionality from those requested by third-party libraries linked with the app. We implement a sophisticated UI automator and conduct an extensive evaluation of our system's performance and find that Reaper introduces negligible overhead, rendering it suitable both for end users (by integrating it in the OS) and for deployment as part of an official app vetting process. Our study on over 5K popular apps demonstrates the large extent to which personally identifiable information is being accessed by libraries and highlights the privacy risks that users face. We find that an impressive 65% of the permissions requested do *not* originate from the core app but are issued by linked third-party libraries, 37.3% of which are used for functionality related to ads, tracking, and analytics. Overall, Reaper enhances the functionality of Android's run time permission model without requiring OS or app modifications, and provides the necessary contextual information that can enable users to selectively deny permissions that are not part of an app's core functionality.

CCS CONCEPTS

• **Security and privacy** → **Mobile platform security**; *Mobile and wireless security*; *Access control*.

KEYWORDS

Android; dynamic analysis; permission origin; personally identifiable information; third-party libraries

ACM Reference Format:

Michalis Diamantaris, Elias P. Papadopoulos, Evangelos P. Markatos, Sotiris Ioannidis, and Jason Polakis. 2019. REAPER: Real-time App Analysis for Augmenting the Android Permission System. In *Ninth ACM Conference on Data and Application Security and Privacy (CODASPY '19)*, March 25–27, 2019, Richardson, TX, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3292006.3300027>

1 INTRODUCTION

Modern smartphones have become a treasure trove of sensitive user data and personally identifiable information (PII) that is regularly collected and exfiltrated by Android applications ("apps") [36, 47, 48, 56, 63, 64]. At the same time, the limitations of Android's permission system have been explored extensively and various modifications have been proposed [31, 47, 65, 72, 75]. The privacy risks that arise from permission management are further exacerbated by the dominating role that third-party libraries have achieved in the Android app ecosystem by providing a revenue stream for developers [54]. On average, 41% of an app's code is contributed by common libraries [49]. The prevalence of libraries has serious implications, as they incentivize apps to request more permissions than needed [70] and extensively leak personal information [2, 4, 5, 52, 62].

Android has been moving toward a more fine-grained permission control system with each major revision, showing considerable improvements over the original design where users were presented with confusing blocks of information at installation time [46]. Following the introduction of the new permission system in Android 6, users can now accept or reject a permission request at run time, or revoke permissions at any time through the system's settings. However, recent work [39] demonstrated that users still do not fully grasp how permissions work and found that they are more likely to deny a permission request when given a detailed description of their personal information that will be accessed and uploaded (e.g., their actual phone number). Lin et al. [50] found that providing users with information on why a resource is being used can alleviate their privacy concerns, while in a different context Wang et al. [78] found that users perceive permissions differently when they are related to an application's core functionality.

Even though the new approach empowers users by enabling a more precise granting of permissions, apps remain a black box with hidden inner workings, thus preventing users from fully benefiting

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CODASPY '19, March 25–27, 2019, Richardson, TX, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6099-9/19/03...\$15.00
<https://doi.org/10.1145/3292006.3300027>

from its potential; as users cannot differentiate between permission requests needed for the core functionality of the app and requests from third-party libraries, they can not make informed decisions regarding which permissions should be granted to each app. Motivated by this rationale, we argue that a fine-grained access control permission system should notify users of the origin of a permission request and explicitly state if it is needed by the app’s core functionality or an integrated third-party library.

To bridge this significant gap we present Reaper, a system for dynamically analyzing apps and inferring the origin of permission-protected calls (PPCs) through inline hooking that enables passive monitoring of the internals of the Android operating system. This requires tackling several challenges, each of which is addressed by one of the main components of our system. First, a dynamic analysis framework requires an efficient tool for traversing the graph of each app with sufficient coverage. We develop UIHarvester, an automation tool that utilizes hooks in the Android rendering process for identifying interactive elements and their properties, for traversing the app’s graph without a priori knowledge of the app’s functionality or visual characteristics. UIHarvester introduces negligible overhead that is 30-38 times smaller than that of Android’s UI Automator, and improves coverage by $\sim 26\%$ compared to the tool that achieved the highest coverage in a comparative study [35].

PermissionHarvester is responsible for the main functionality of Reaper, as it hooks PPCs at run time and extracts the current stacktrace. Since the permissions required by functions are not the same across all Android versions, with newer versions not requiring permissions for certain calls that access PII, our tool automatically recognizes the OS version and adjusts its functionality accordingly. Even though PPCs protect device resources, common users do not have complete knowledge of Android’s documentation and internals and are mainly concerned with apps accessing personally identifiable information. As such, PermissionHarvester also monitors PII access regardless of whether the call is protected by a permission or not. Extracted stacktraces are processed for identifying the origin of calls that are protected by permissions or access PII. Our approach is not affected by encryption techniques that may attempt to hide the presence of third-party libraries and the exfiltration of PII. Furthermore, Reaper does not require any modifications to the OS and does not depend on any sort of instrumentation, thus, introducing minimal performance overhead (we only require root access). Our system can be incorporated as part of the Android Open Source Project for enriching the contextual information shown to users.

To explore the potential benefits of our system, we use Reaper to analyze over 5K popular Android apps, and find several alarming results regarding the extent of third-party libraries’ use of permissions and permission-protected calls. Indicatively, our study reveals that for 90% of the apps third-parties initiate more permission protected calls than the core app itself. We find that on average 65% of used permissions are needed by third-party libraries, and 34% of the apps never issue PPCs from their core code as the requested permissions originate solely from library code. To make matters worse, when it comes to *dangerous permissions* 48-59% of the requests originate from third-party libraries. For permission-protected calls that reach PII, in 59% of the apps third-party libraries use `getRunningAppProcesses()`, which can lead to precise user

identification [23]. We also find many libraries accessing PII from non-permission protected calls. Finally, we explore how the origin information is augmented by accounting for the library type. We find that at least 37.3% of PPCs and 28.6% of PII accesses that originate from libraries are exclusively intended for functionality related to ads, tracking and analytics and could safely be denied by users without preventing the apps’ intended functionality.

The key contributions of our work are the following:

- We develop Reaper, a **real-time permission analysis** system that infers the origin of calls to permission-protected resources or non-permission-protected sensitive PII. Our system can augment Android’s run time permission system by enriching the contextual information provided to users.
- We experimentally evaluate our system and find that the overhead introduced is minimal, rendering it suitable for analyzing apps at a large-scale, or integrating in user devices.
- We use Reaper to explore the interaction between libraries and Android’s permission system in depth. We provide a fine-grained analysis of PPCs and PII access *by third-party libraries* in the wild. Our findings shed light on the alarming extent to which libraries dominate such calls, and motivate the need for incorporating origin information in permission requests.
- We publicly released our source code and the datasets used at <http://www.reaper.gr>.

2 BACKGROUND AND MOTIVATION

The incorporation of third-party libraries allows app developers to take advantage of useful existing functionality and also tap into an alternative revenue stream without the need to charge users for the app itself. While this may appear beneficial to end users as they are able to obtain apps seemingly for free, it suffers from the inherent privacy risks of the prevailing paradigm of services being free because users are the product [3] and “pay” with their personal data [54]. Not only have such libraries become prevalent (49% of apps contain at least one ad library [57]), but the risks they present [44] increase through time as they ask for increasingly more dangerous permissions [32]. This necessitates the deployment of functionality that can differentiate between permissions required by the actual app and those requested by third-party libraries. Tracing permission requests back to third-party libraries allows for enriching the contextual information presented to users.

While libraries can offer useful functionality to app developers, they may also surreptitiously add (potentially dangerous) permissions without the developer being explicitly informed. As such, users cannot rely on developers’ intentions or safe practices for ensuring appropriate access to their data. Indeed, Android supports the merging of multiple manifest files, as each APK file can contain only one manifest file. While this functionality is meant to facilitate the inclusion of external libraries, it also allows third-party libraries to silently include permissions without the developer’s approval. To make matters worse, developers have to explicitly and proactively include specific commands (`tools:node="remove"`) in their manifest to prevent libraries from including specific permissions.

To verify that this occurs in practice, we conduct an exploratory experiment with popular libraries. We create a test app and separately integrate each library; after compilation we extract the final

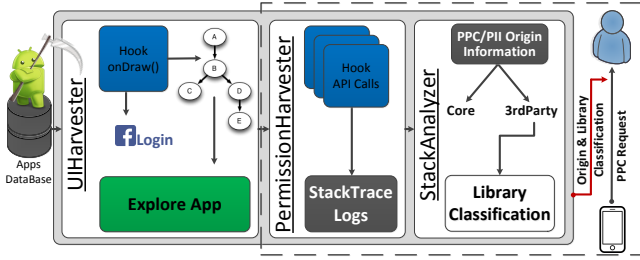


Figure 1: Overview of Reaper’s architecture.

manifest file to see which permissions have been included by the libraries without any form of notification. First, we look at one of the most prevalent libraries [18], namely Google Play Services. Google offers multiple libraries and we test Firebase and GMS (which incorporate functionality for analytics, ads etc.) and find that they add six and eight permissions respectively. We investigate whether libraries also merge dangerous permissions, and find that Instabug and PayPal silently include three (read, write access to External Storage and Record Audio) and one (Camera) dangerous permissions respectively. This simple experiment highlights how Android offers functionality that can be misused by third-party libraries to silently obtain access to permissions that can affect the user’s privacy or the device’s normal operation. It is not meant to be exhaustive and many more libraries may be exhibiting such behavior in practice.

3 REAPER DESIGN AND IMPLEMENTATION

Reaper’s primary goal is to distinguish which permissions are needed by the core functionality of the app and which by integrated third-party libraries. Many advertising and tracking companies freely provide preconfigured libraries and online tutorials on how to integrate them. Moreover, previous work [44] found that advertising libraries are prone to downloading code over HTTPS and dynamically loading and executing this code using the `DexClassLoader` class. Even though dynamic code loading offers useful functionality to developers, it can also be used to evade static analysis [59]. Furthermore apps may also hide their functionality through common obfuscation and encryption techniques [11, 36].

We leverage the hooking mechanism of Xposed [1] to build a dynamic analysis system that is designed to overcome the aforementioned obstacles. We require root access but do not rely on any OS modification, such as changing and recompiling the AOSP image, allowing us to apply it to any stock Android version. Figure 1 shows an overview of Reaper, which consists of three components: (i) UIHarvester (Section 3.1) a sophisticated UI automation tool for exercising apps, (ii) PermissionHarvester (Section 3.2) for hooking and monitoring the stacktrace of functions that lead to permission checks, and (iii) StackAnalyzer (Section 3.3) for analyzing stacktraces and inferring if they originate from a third-party library and of what type. If our system is adopted as part of an official app vetting process, or used by other researchers for dynamically analyzing apps, then all three components should be used, as shown in the gray box. If Reaper’s functionality is integrated in the OS for augmenting the permission system, then only two of those components are required, as shown in the dotted line.

3.1 UI Harvester

A significant challenge when performing dynamic analysis on mobile apps is the traversal of the app’s graph through the simulation of user interactions, without any a priori knowledge of the interactive content that will be displayed in the app. Previous work [24, 27, 33, 45, 53, 60, 61, 71, 74, 81, 85, 86], has explored the dynamic traversal of an application from different perspectives, such as achieving high traversing coverage or identifying malicious behavior. Unfortunately apart from requiring static analysis of the apk [24, 27, 45, 60, 71, 74, 81, 85], they may require some form of app instrumentation [27, 45, 60], or OS code modification [53, 61, 85], or are pinned to a specific Android version [24, 33, 85, 86].

UI Automator [21] is a useful tool available from the Android SDK that offers functionality similar to what we require; it can dump the interactable objects of the display and provide additional information about them. However, UI Automator presents two major disadvantages that render it unsuitable for our needs. First, if the app uses the `WindowManager.LayoutParams.FLAG_SECURE` option, UI Automator has to respect this specific flag and cannot output information about the objects being displayed. This flag is a security feature that treats the contents of the window as “secure” and prevents taking screenshots or being viewed on non secure displays [22]. This flag is not uncommon and is used to secure apps (e.g., PayPal) from side channel attacks, and can be used by apps or third-party libraries that want to evade dynamic analysis. Second, the performance overhead introduced is significant, rendering UI Automator unsuitable for a large scale analysis (details in Section 5).

To overcome the aforementioned restrictions, we developed a **plug & play** prototype that simulates user interactions, which fulfils the following design constraints:

- No requirement for a priori information on the content that will be displayed in the app.
- No requirement for decompilation or static analysis of the apk file, or access to the app’s source code.
- No requirement for code modification (app or OS).
- No inefficient and ineffective random input generation approaches.
- Support for a wide range of Android versions.

Harvesting UI elements. Android renders the contents of the display based on a specific procedure, where each activity receiving focus provides the root node of the layout hierarchy and draws its layout. Drawing starts at the root node of the layout tree and is traversed in a top-down order. Android also provides the `View` class, which is the basic building block for UI elements. While traversing the tree each `View` is rendered in the appropriate region. Rendering begins with a measure pass and continues with a layout pass. The former, is responsible for the dimension specifications of each `View`, while in the latter each parent positions all the children based on the measurements obtained during the measure pass. After this two-step process, the `onDraw()` function of the `Canvas` is called for rendering the contents of a `View` object. Whenever something changes on the display, `View` notifies the system and, depending on the changed properties, either the `invalidate()` or the `requestLayout()` functions are called. The former calls the `onDraw()` for the specified object while the latter instantiates the procedure from the beginning. Since `onDraw()` is called last before the actual rendering, we hook it to capture the displayed elements.

Identifying interactable objects. Having access to a View object enables us to use every method of the View class [14] from the Android SDK. This allows us to detect what type of elements are contained in the View object (e.g., TextView, Button, Image), as well as the corresponding metadata such as the text displayed, the resource-id, the index, and horizontal/vertical scrolling. To identify whether the elements of the View object are user interactable, we use the `getImportantForAccessibility()`, `hasWindowFocus()` and `isShown()` methods on each object. Moreover, we also find the position and the coordinates of each object by using the methods `getLocationInWindow()` and `getLocationOnScreen()`.

Traversing applications. UIHarvester hooks into the `onDraw()` function and exports all the information to logcat. Using a Python parser we extract this information and use all the interactable objects for performing a breadth-first traversal of the app. Since we know the type of each element as well as its coordinates, we utilize the Android Debug Bridge for performing the appropriate actions (e.g., tap, swipe, keyevents, etc.). Every time a new View is drawn on the display (e.g., after a button is pressed), UIHarvester exports its elements. By obtaining all the Views that have been drawn on the display, we can recreate the app's UI graph.

Login. Many apps provide a login option for a more personalized experience, while others require users to login before using the app. Not being able to handle such cases significantly limits the coverage and usefulness of any UI automator. As such, we implement an automated login feature that leverages Facebook's SSO.

Setting a threshold. Due to the dynamic nature and content of Android apps, it is possible for UI automators to get stuck in an infinite recursion of state transitions for certain apps. A simple example is the "back to menu" button. We handle this case by checking whether the elements (and their properties) have already been encountered in the exact same order. A case that can not be handled by our approach is when an activity renders content that is downloaded from the web and changes between transitions. As such, we need to impose a threshold for terminating the traversal of these apps. Since our goal is to execute as many permission-protected functions as possible, we set a rule to stop the traversal when five minutes pass from when the last permission request occurred. While this could potentially result in a loss of certain requests, tracking and advertising libraries often perform their functionality either at launch time or after the user logs in [7, 55].

3.2 Permission Harvester

This is the core component of Reaper and is responsible for monitoring function calls and logging the current stacktrace for subsequent analysis. Since blindly hooking into every function call would result in an increase of the overhead without providing any additional information, we first need to identify the functions that lead to a permission request from the Android Server.

Permission mappings. The constant evolution of Android's permission system has led to many changes as well as compatibility and security issues regarding how each permission works [84]. Due to the differences between API versions, functions may lead to other permissions or may no longer be permission-protected across versions. For instance, the `getScanResults()` function from the `net.wifi.WifiManager` class only needs the `ACCESS_WIFI_STATE`

permission up to API 22. Since API 23, this method also requires access to the device's location through `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION`. The Stowaway project [41] used static analysis in Android 2.2 to determine an app's API calls, and provided a map that identifies what permissions are needed for each API call. Recently PScout [26] and AXPLORER [29], statically analyzed the Android Framework, extended the mappings for newer versions and corrected previous uncertainties. When comparing the mappings of PScout and AXPLORER, we find various differences in their results; in API 22 AXPLORER registers the function `getWifiState()` in `net.wifi.WifiManager` with the `ACCESS_WIFI_STATE` permission. On the contrary, PScout registers the same function with the `DUMP` permission. As such, it is important to dynamically validate the permission mappings as we discuss below.

Mappings selection. Our system identifies the OS version and adjusts the permission mappings, using AXPLORER's results for APIs 16 (Android v. 4.1) to 25 (v. 7.1) by hooking the appropriate functions. We excluded API 20 as AXPLORER does not provide mappings for this version. To facilitate our system's description we will refer to an example permission-protected call and the induced hooks as illustrated in Figure 2. By monitoring the PPCs ①, we can identify the corresponding permission through AXPLORER's list and the origin of the function call through the stacktrace of the current thread. The Java stacktrace holds every execution until a Binder transaction occurs, and also reveals the path and exact Java file (inside the apk) from which the call originated ②.

Validating permission mappings. To dynamically validate the mappings from previous work, we need to hook the appropriate functions of the Android Server. The `checkPermission()` and `checkPermissionWithToken()` functions (found in the class `ActivityManagerService`) grant or deny access to resources according to the app's permissions. Prior to API 22 access to these two functions is feasible by directly hooking them. Since API 22, a different entry point is needed for reaching them. To reach the methods and classes of the Android framework we hook the `systemMain()` function of the app's `ActivityThread` class. Within that hook we can monitor the permissions that each app and process request at run time by encapsulating the hooks for these functions.

Handling asynchronous calls. In Android different resources are handled by different System Services. For an app to access such information ①, a new thread of the appropriate service manager is created and this newly created thread calls the validation check mechanism ②. During this process, Android Binder is responsible for passing messages between entities, using the `onTransact()` and `execTransact()` functions. Even though the functions involved in permission validation are asynchronous calls, we know a priori the functions that will lead to a permission request ① and can call them sequentially and map each function call with the appropriate permission check that occurs on the Android Server ②. To this end we created a mock application that executes in a sequential manner all the permission-protected calls of the Android SDK.

In practice asynchronous callbacks are frequently used in Android, and a library can register its functionality, or part of it, as a callback. Even though the registered callback executes in a separate thread, the stacktrace of this newly created thread contains the origin of the embedded executed code. Since PermissionHarvester monitors the execution of PPCs independently of asynchronous

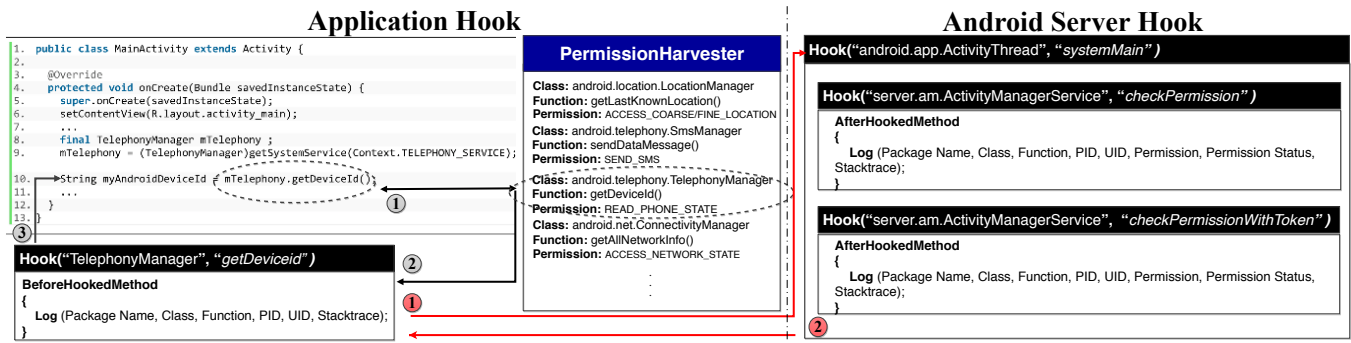


Figure 2: Application Hook is PermissionHarvester’s core hooking mechanism that monitors PPCs and inspects stacktraces for extracting their origin. Android Server Hook is only used for validating the permission mappings.

```

0 java.lang.Thread.getStackTrace(Thread.java:580)
1 android.location.LocationManager.getLastKnownLocation()
2 com.appodeal.ads.an.e(SourceFile:243)
3 com.appodeal.ads.d.b.<init>(SourceFile:180)
4 com.appodeal.ads.d.i.a(SourceFile:295)
5 com.appodeal.ads.d.i.a(SourceFile:105)
6 com.appodeal.ads.d.i.doInBackground(SourceFile:37)
7 android.os.AsyncTask$2.call(AsyncTask.java:292)
...
13 java.lang.Thread.run(Thread.java:818)

```

Listing 1: Example stacktrace for getLastKnownLocation(). The code that initiated the PPC through an asynchronous call belongs to "com.appodeal" package, which corresponds to the Appodeal third-party library.

calls, the StackAnalyzer component can identify the true origin of the PPC. We illustrate this process with an example of a library registering a PPC in an asynchronous callback. Listing 1 presents a callback that executes the `getLastKnownLocation()` function needed by the "com.appodeal" library. This library creates a subclass of an `AsyncTask` and overrides the method `doInBackground()`. Even though the code that is placed in this method is executed in a different thread, thus obscuring whether the core app or the library registered the callback, we can still successfully identify whether the executed code belongs to a third-party library.

Non-permission-protected PII leaks. It is important to note that not all PII is protected by permissions, and library developers may take extra measures to hide the presence of PII leaks and the surreptitious exfiltration of data (i.e., obfuscation, encryption and dynamic code loading). As PII can enable user tracking, it is crucial to identify the origin of such requests. Recent work [56] released an extensive list with such device characteristics that are leaked. We manually map those characteristics with their appropriate function calls and find that 8 such functions are not permission-protected. By extending Reaper to support these calls, our system is able to identify the origin of PII leaks regardless of the call being protected by a permission or not. While this is not part of our work’s main focus, we include this information to further highlight the invasive behavior of third-party libraries in our study in Section 6.

Our approach can reveal privacy leakage without the need to perform deep packet inspection, thus, bypassing the obstacle of attempting to identify data exfiltrated in an obfuscated form, which

has stifled previous work on PII leakage. For instance, in our experiments we found two apps (com.sevideo.slideshow.videoeditor, com.fourvideo.videoshow.videoslide) that integrate the XavirAd library, which downloads a dex file from a remote server, collects PII and sends them encrypted over the network [11].

3.3 Stack Analyzer

Apart from being useful for debugging, stacktraces can be used during run time execution since they contain essential information about the current thread. We opted for this approach as it provides a straightforward solution for identifying the origin of a function. The stacktrace contains a path to the source file and has four fields of interest: the package name, the class from which the method was called, the actual method and the file name of the source code. StackAnalyzer processes the stacktraces of important calls and checks if the package name of a known third-party library exists in the path of the stacktrace’s function call. Even though library code can be obfuscated (e.g., classes, functions, etc.), by default library package names remain intact since developers need to know which library to link in their app during the build process. To verify that stack inspection is effective in practice, we manually examined the package name of all the stacktraces collected from our experiments (see Section 4) and found that only 1.14% have an obfuscated package name, preventing us from identifying their origin. This is further corroborated by Wermkeet al. [79], who conducted an obfuscation detection analysis on 1.7 million apps from Google Play and found that even for obfuscated libraries larger scopes remain identifiable in package names (e.g., com.google.ads.*).

Third-party library package names. Li et al. [49] conducted a large scale analysis of 1.5 million apps from Google Play, in order to identify common Android libraries. Even though their approach does not handle obfuscated code, they identified 1,353 third-party libraries. LibScout [28] bypassed the limitations of obfuscated code by using a variant of Merkle trees and performing profile-matching between known third-party libraries and the contents of the apk file being tested. Since the results provided by LibScout are bound by the dataset they are trained with, it is possible for LibScout to miss some of the libraries that are integrated in the application. Indeed, during our experiments we came across such an example: AppsFlyer [15] a well known mobile tracking library. StackAnalyzer

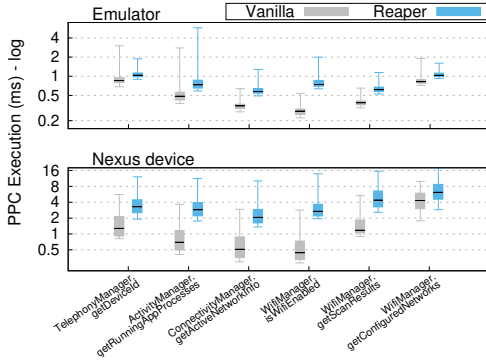


Figure 3: Performance overhead of PermissionHarvester, including the overhead for the hook.

uses the combined results of both systems to create a coherent list of package names and to identify at runtime whether the stacktrace belongs to code originating from a third-party library.

Library classification. In practice, developers may use code from a third-party library that is integral to the app’s functionality. By classifying the type of the library from which the permission request originates, we obtain more detailed information regarding the nature of the call and whether it can be attributed to code that is necessary for the app’s intended functionality; e.g., by differentiating calls from an ad library to those from an app-development library. By disambiguating the origin of the calls Reaper further augments the contextual information presented to users and guides them towards granting “useful” permissions. Specifically, our system uses information from two sources [13, 49] to ascertain the category of the library from which each third-party call is initiated at runtime and provide that information to the user.

4 DATASET & EXPERIMENTAL SETUP

We downloaded free apps from Google Play using Raccoon [20] and performed the majority of experiments using emulators. We opted for an emulator for the ability to deploy multiple virtual machines and analyze a large amount of apps. While third-party libraries may be able to infer the presence of a virtualized execution environment and alter their behavior, previous work on app analysis has also relied on emulators [68, 69]. To make the environment look more like an actual device we installed the Google Play services and signed in with a legitimate Google account. We conduct the experiments in Android API 22, as it is the API with the most accurate permission mappings available – AXPLORER’s mappings for API 23 are incomplete [17]. Overall, we selected the top 300 apps (or as many as were available) from each category, and downloaded a total of 5457 from 38 categories.

5 PERFORMANCE EVALUATION

Here we evaluate our system’s performance and measure the overhead introduced by each of the main components. We also compare UIHarvester to popular tools and demonstrate the advantages of our approach. We perform experiments using both an emulator and a Google Nexus 6 device, running the AOSP image with API 22.

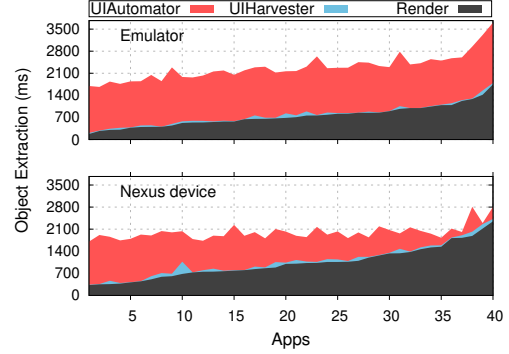


Figure 4: Performance overhead comparison between Reaper’s UIHarvester and UIAutomator.

PermissionHarvester overhead. The same code handles every PPC. Using a mock app that individually issues six PPCs from different managers, we measure the time needed for each PPC with and without PermissionHarvester present. In the vast majority of cases the function calls tested had an execution time of less than 1ms and 4ms for the emulator and the real device respectively. Since `System.currentTimeMillis()` does not produce readings of less than 1ms, we used the `System.nanoTime()` to extract a more accurate representation of the execution time. Figure 3 presents the results from 50K executions of the app. We observe that even though the same code applies to every hooked PPC, the induced penalty varies between 0.18-0.45ms for the emulator and 1.93-3.77ms for the Nexus device. The reason for this is that each PPC can result in stacktraces of different sizes. While `System.nanoTime()` is significantly more accurate than `System.currentTimeMillis()`, it is a relatively expensive call. It depends on the underlying architecture and can take up to 100 CPU cycles while measuring with millisecond precision takes only 5-6 CPU cycles. Apart from being more expensive, it also exhibits deviation in its execution time, which is reflected in the larger deviation of `getRunningProcess()` and `getDeviceID()`. Overall, the overhead for the actual hook is 0.0075ms [6] and the remaining overhead is due to the system call required for logging the stacktrace.

UIHarvester overhead. We measure the induced overhead of UIHarvester, by checking the extra time needed to render the contents of the display. We use the “Displayed” value from logcat, which represents the time elapsed between launching an activity and drawing its contents. For this experiment we selected 40 apps of different sizes and varying loading times, and measured the time needed to launch the main activity with and without UIHarvester.

As shown in Figure 4 the penalty in the emulator is between 0.3%-21% with an average of 6.55%, and depends on the number of elements drawn in the display. In the device the penalty is 0.16%-56.59% with an average penalty of 7.8%. In the worst case, the overhead to render the contents of a heavy View is 140ms for the emulator and 386.1ms for the device, which is acceptable for fully automated dynamic analysis. We also compare to the time needed to extract information about the display using UIAutomator, which offers similar functionality. On average UIHarvester only requires 40.19ms for the emulator and 67.29ms for the device. UIAutomator

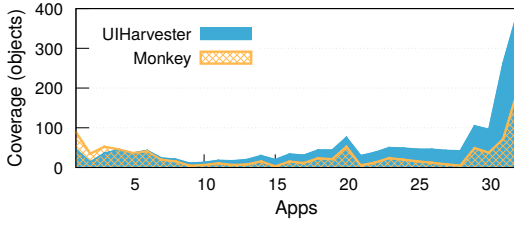


Figure 5: Comparison of interactable object coverage between Reaper’s UIHarvester and Android’s Monkey.

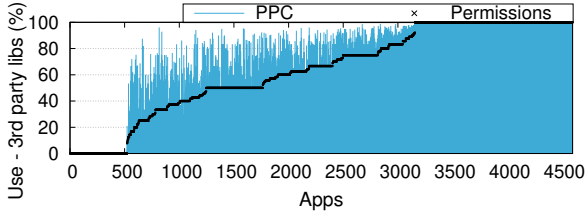


Figure 6: Per app use of permissions and PPC by libs.

takes over 1,546ms and 2,001ms to extract the elements respectively, resulting in a 30 to 38-fold increase. Thus our tool offers superior performance while being more effective for this study.

UIHarvester Coverage. While GUI exploration and coverage in Android can be measured using different techniques (line coverage, activities, crashes, etc.) we opted for counting the interactive elements since it can be applied to both open and closed-source apps without the need for instrumentation. Choudhary et al. [35] evaluated six input generation techniques and compared them to Android’s Monkey. They found that the Monkey fuzzer was the best option, achieving a 40% coverage. To evaluate UIHarvester’s coverage, we obtained the same set of apps from [35] and compared the interactive objects found by UIHarvester and Monkey; we tested the 32 apps that remain functional. Since Monkey can only perform random clicks and does not count the interactive elements, we used the technique employed in UIHarvester to extract them. For a direct comparison, we set a timeout of 5 minutes and configured the time required to generate input events to be consistent between both tools. In Figure 5 we plot the number of objects found by Monkey (averaged over three runs) and the objects found by UIHarvester. Overall, UIHarvester improves coverage by 25.98%.

Compatibility between versions. A common limitation of analysis tools is being pinned to a specific Android version. By designing Reaper to have minimum dependencies, we maintain compatibility across APIs. We verified this by analyzing ten backwards-compatible apps on the four most common Android versions [12] and found that all Reaper components remained fully functional.

6 PERMISSION ANALYSIS

We study 5457 apps in order to understand the use of PPC and access of PII by third-party libraries in practice. Our study dynamically examines the origin of such calls, enabling a fine-grained exploration of the corresponding privacy risks.

Table 1: Issues that resulted in certain apps not being traversed during our experiments.

Apps Without Interaction - Addon (41), Launcher (31), Plugin (9), Theme (5), Widget (3)	89
Manual Login Required	45
Installation Failure	37
Device Specific	15
Emulator Detection	1
Root Detection	1
Malfunction / Crash	156
Total	344

Apps without PPC. In our experiments 315 apps did not issue a PPC call during their analysis. Furthermore, we were not able to traverse an additional 344 apps and obtain a PPC stacktrace. Table 1 breaks down the numbers for the issues that resulted in this; 89 apps could not be traversed due to their type, as they do not contain launchable activities and there is no direct interaction. Out of the remaining, 45 required a manual login, 37 failed during installation, 15 apps were for a specific device brand or only available for certain CPUs/GPUs and 2 apps did not execute because of the device’s environment. Also, 156 apps malfunctioned at launch time. To understand whether Reaper affects these 156 apps, we tested them without our framework, and observed that in both cases the apps remained non-functional. When executed without the Xposed framework, 155 apps continued to crash. While one app appears to be detecting Xposed, this can be trivially bypassed by renaming the Xposed package. Thus, practically, our experimental environment only prevented one app from running. To analyze apps that perform emulation or root detection, Reaper can also be used with a real device and the root requirement can be hidden using known root-hiding techniques [9]. Interestingly, certain apps that can not be traversed due to their type still perform PPCs (at launch time).

Third-party library use. In Figure 6 we explore the use of PPCs and their corresponding permissions by libraries. We observe that for 521 apps PPCs are only used by the app’s core functionality, while for 1,642 apps every PPC originates from third-party libraries. While there is varying behavior in the remaining 2,635 apps, there is significant use of PPC throughout. Overall, 65.22% of the permissions requested are not from the apps’ core code, but are requested by the libraries. These results verify our intuition that PPCs and their underlying permissions are heavily used by third-party libraries, with 34% of these apps never calling them from their core code. This highlights the benefit of adopting the functionality offered by Reaper for informing users about the origin of permission requests and enabling more fine-grained control.

Function and permission origin. To better understand the origin of each function, we explore their use across all apps. As shown in Figure 7, use of permission-protected functions by libraries remains high and certain functions *are never used by core functionality*. For instance, one such function that also accesses PII, is `getSubscriberId()` which returns the device’s IMSI.

In Figure 8 we plot the 30 most used functions. We find that these typically are calls that return device specific information, such as Device-IDs, Network-Info, SSIDs, Location, Apps-installed, which

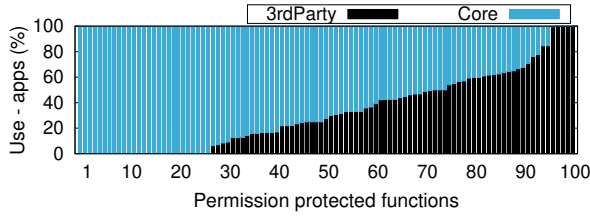


Figure 7: Origin of every permission protected function.

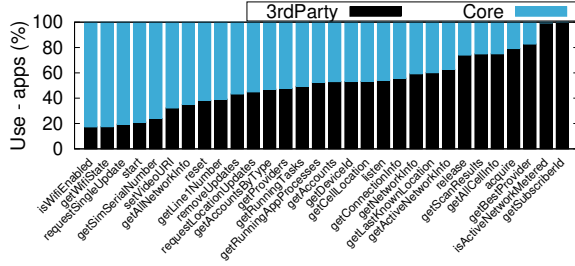


Figure 8: Breakdown for the 30 most used PPCs.

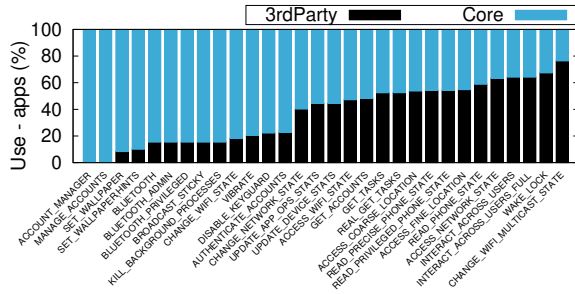


Figure 9: Breakdown for the 30 most used permissions.

are considered PII and are used by advertising and tracking companies. We observe that the `getSubscriberId()` function which is also included in the top 30, requires the `READ_PHONE_STATE` permission which is one of the permissions considered dangerous by the Android developer guide. In Figure 9, we plot the 30 most used permissions. We manually mapped these permissions to their protection level from the official Android source code [19] and found that third-party libraries also use permissions that fall in different protection levels such as signature, privileged, installer, development and dangerous. The use of the four dangerous permissions (i.e., `ACCESS_COARSE_LOCATION`, `ACCESS_FINE_LOCATION`, `GET_PHONE_STATE`, `READ_ACCOUNTS`) ranges from 48% to 59% for third-party libraries. This means that for these apps *when users are presented with a dangerous permission request at run time, roughly half the time the permission does not originate from core code.*

Third-party library integration. To understand how many third-party libraries are used inside apps, we calculate the fraction of distinct third-party libraries using PPCs, as well as the total fraction of PPCs attributed to 3rd parties or core functionality. In Figure 10 (left) we observe that 30% of our dataset contains at least two distinct third-party libraries that initiate PPCs. Moreover, as

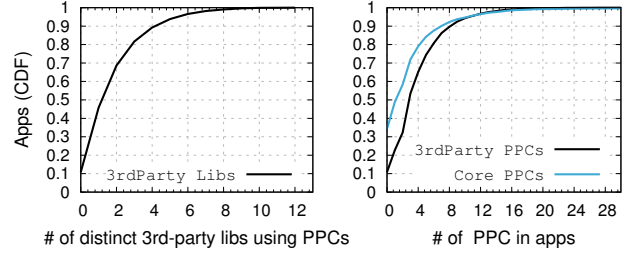


Figure 10: Distinct libraries issuing PPCs in each app, and breakdown of PPCs for third-party and core functionality.

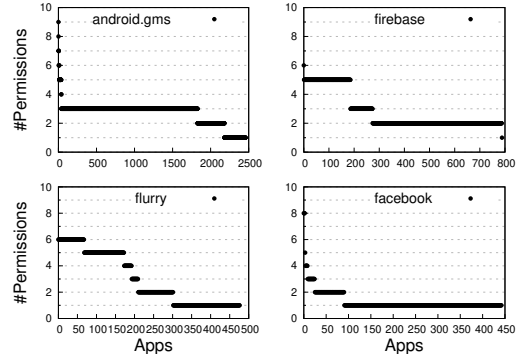


Figure 11: Number of permissions used across the apps that include the 4 most used libraries.

can be seen in Figure 10 (right), for 90% of the apps third parties initiate more PPCs than the app's core code.

Permission variation. We found cases where a library uses a different number of permissions across apps. In Figure 11, we select four of the most used libraries and plot the number of permissions used across all apps. One possible reason for this could be because UIHarvester was not able to reach a level of coverage that would trigger all the permission requests. However this will not always be the case since apps contain different versions of the same library, which may offer different functionality. Furthermore, libraries may also adjust according to the number of permissions granted [16].

PII access. While not our main focus, an important aspect of our analysis is exploring the extent of third-party libraries accessing PII; the origin information provided by Reaper results in a more fine-grained and precise view of PII leakage when compared to prior studies that explore apps' behaviors as a whole. We map function calls to PII based on the identifiers provided by prior work [56, 63] and the Android SDK documentation, and analyze the information provided by Reaper. Figure 12 shows all the functions that access PII, whether through a permission-protected call (blue circles) or not (red circles). The size of the circle denotes the number of apps that contain the respective library and issue the corresponding function call. Due to space constraints we only include the 13 most popular libraries. We find that third-party libraries access the non-protected calls more frequently than the permission-protected calls. As users can be fingerprinted from the information returned by these functions, it is troubling that Android

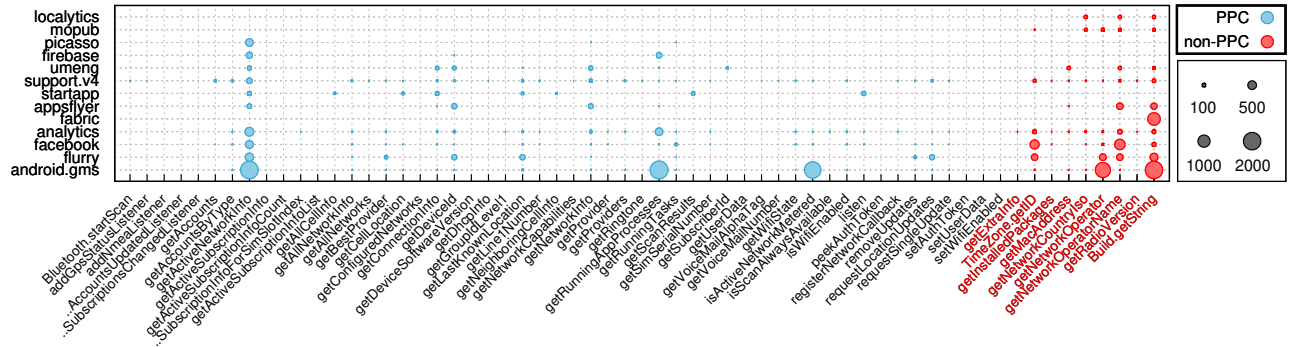


Figure 12: PII leakage from the most popular third-party libraries (sorted in descending order) broken down to the corresponding function call used. Blue circles denote PII being accessed through permission-protected calls, while the red circles indicate PII access by functions that are not permission-protected. The size of the circle denotes the number of apps in each case.

does not enforce a permission requirement. *We argue that all calls that lead to PII should be permission-protected, allowing users to manage what information can be accessed by apps and third parties.* The `getRunningAppProcesses()` function can also be called without the `GET_TASKS` permission. It returns a list of running processes and is being used by 3,218 (59%) apps. This is also a significant privacy threat, as previous work has shown that tracking companies can potentially identify users when as few as 4 apps are known [23]. Despite being discontinued for APIs ≥ 23 , it remains active for older versions, with 66.4% of devices running APIs ≥ 23 [12].

Library classification. To further enrich the origin information, Reaper classifies the type of libraries initiating each monitored call, which allows our system to further disambiguate the origin of calls. In Figure 13 (left) we present the coarse-grained classification of the type of library from which each call originated. Libraries that have multiple labels are counted in all respective categories. For a subset of the libraries we also obtain fine-grained information regarding the functionality that they offer, which we present in Figure 13 (right). While we provide a coarse-grained classification of all the calls initiated by libraries in our dataset, we are not able to obtain fine-grained labels for all the libraries identified in our experiments. Specifically, we obtain labels for 84 out of the 234 libraries that issue PPC calls, and for 75 of the 203 third-party libraries that access PII. As can be seen in Figure 13 (right), analytics-based libraries are responsible for the most PPC calls and PII accesses, while ad-related libraries issue more calls when the different subcategories are aggregated. It is also evident that specific libraries that ease the app development process are very common.

Using the coarse labels, we find that 15,610 PPCs and 21,322 PII accesses originate from libraries that are *exclusively* labelled as developer libraries, indicating that they are needed for the app’s core functionality and should likely be granted. On the other hand, 1,287 PPCs and 845 PII accesses originate from tracking or ad libraries, and can be safely denied. Furthermore, for libraries with multiple coarse-grained labels, we leverage the fine-grained labels and find that an additional 9,129 PPCs and 5,240 PII calls can be safely denied as they are used exclusively for analytics and advertising. Three of the most used libraries (facebook, google.gms and firebase) cannot be excluded using fine-grained labels as they cover a wide spectrum

of functionality and contain numerous labels; however, for all three we can infer which aspect of their functionality is used in each call as the respective package name (e.g., `com.google.android.gms.ads`) explicitly denotes it (obviously, this approach cannot be applied to untrusted or unknown libraries). As such, the stacktraces allow us to identify an additional 10,424 PPCs and 11,580 PII calls than can be denied as they are used for analytics, ads, and tracking.

Overall, out of the 55,859 distinct PPC calls Reaper would enable users to safely deny 20,840 (37.3%) PPCs without preventing apps from leveraging third-party code for core functionality. Similarly, out of 61,602 PII accesses users could safely deny 17,665 (28.6%). Thus, apart from augmenting the permission system by providing rich contextual information, Reaper can further help users by providing concrete recommendations to accept or deny a considerable number of “straightforward” permission requests. The information provided by our system can also be used to expand access control tools like XPrivacy, allowing for more fine-grained control of user data, as it can selectively block invasive calls originating from third-parties while allowing such calls required for core functionality; we consider this part of our future work. For the remaining calls, displaying the library’s type and the specific permission requested can significantly improve the existing permission system and better guide users into making informed decisions based on the app’s intended functionality.

Permission mapping inconsistencies. While Reaper relies on permission mappings provided by prior work, our system can be used to dynamically validate those statically generated mappings. Thus, while not part of our study’s main focus, we conduct an exploratory study as more accurate mappings will further improve the main functionality of our system; we opt for API 22, since it is the most recent version with the most accurate permission mappings. We created a mock application that sequentially executes all the permission-protected calls of the Android SDK, and verified the permission of each function call based on the permission check occurring in the Android Server. The `divideMessage()` function exists in two different classes, and PScout and AXPLORER report different permissions for this function. Using Reaper we found that this function does not need a permission. To further verify this result, we triggered this function without declaring any permission in

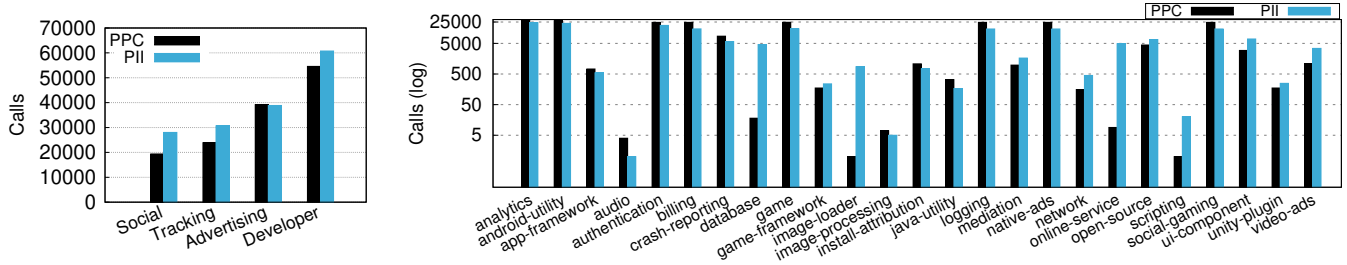


Figure 13: Coarse-grained (left) and Fine-grained (right) classification of PPCs and PII accesses initiated by libraries.

the app’s manifest file and observed the same functionality without any warnings, errors, or crashes. We also manually investigated certain functions that were not mentioned in either study, and found that prior work missed `sendStickyBroadcast()`, which requires the `BROADCAST_STICKY` permission.

Interestingly, we find that functions being permission-protected also depends on the arguments provided. For example, the function `getprovider()` from the `LocationManager` class is permission-protected when provided with `GPS_PROVIDER` but does not need a permission for `KEY_LOCATION_CHANGED`. We argue that there is dire need for better documentation of the internals of Android permissions, as such scenarios can further confuse developers.

7 DISCUSSION AND LIMITATIONS

Defining origin. Reaper distinguishes core from third-party functionality based on the origin of the executed code. We compiled a list of libraries using data from [8, 28] for identifying their origin. If a library is not in the list, the corresponding stacktrace will not be flagged as third party functionality. Similarly, our library classification relies on external resources [8, 13]. As lists of libraries are extended and become more complete, Reaper’s coverage will also increase. Moreover, core functionality could potentially be misclassified if a function has the same name with that of a known library. We investigated our dataset and did not find such instances.

Call mappings. Reaper maintains a permissions-to-function-calls mapping that contains the functions that should be monitored. While we have used Reaper to validate these mappings, expanding the mappings reported by prior work is out of our scope; thus, our system will not monitor PPCs missing from that list.

Native code. Android apps are written in Java or in native code. Xposed is able to hook functions written in Java, as well as native code in cases of JNI. However, we cannot hook custom native code written by developers since it is not supported by Xposed.

Kernel permissions. Certain Android permissions are regulated by the kernel. Since Pscout and AXPLORER did not conduct a native code analysis and have not created mappings for such permissions, we have not included these permissions in our study.

Graph coverage. UIHarvester may miss displayed content when apps use wrappers or webviews, as will UI Automator.

Emulators. Since apps or libraries can identify that they are being executed in a virtual environment [58], our results may present a lower bound of the privacy risks posed by libraries.

Obfuscated package names. While PPCs that originate from obfuscated package names only account for 1.14% in our study, Reaper could incorporate a static analysis tool like LibScout to reverse the obfuscated package name back to its original form.

8 RELATED WORK

A plethora of prior work has explored the Android operating system in depth. While previous studies have not explored the origin of PPCs and PII accesses and how it can be leveraged to augment the permission systems, they have explored complimentary directions including data leakage and the dynamic analysis of apps. Due to space constraints we only present the most relevant prior work, and discuss how Reaper improves upon, or compliments, that work.

Leak detection and prevention. Meng et al. [54], studied the privacy concerns that arise from in-app advertising, and found that ad publishers can identify user demographic information. Son et al. [67] studied the isolation of different Ad SDKs, and showed that the same-origin policy is not sufficient for protecting users’ privacy. Ad libraries also have the potential for increased data collection through side-channels [38]. Papadopoulos et al. [56] analyzed what leaks occur while accessing the same service through the mobile app and a mobile browser, and showed that accessing a service through the app leaks more device-specific information. Agrigento [36] is a black-box differential analysis tool capable of identifying leaks even in the presence of obfuscation offering significant improvement over prior work. However, their approach cannot handle apps that use custom encryption or custom or native code for certificate checking. On the contrary, Reaper can detect when an app attempts to access such data, and could be used in conjunction with their system. Moreover, their differential analysis could be improved by incorporating UIHarvester for exercising apps.

FLEXDROID [65] is an extension to Android’s permission system that provides dynamic, fine-grained access control for third-party libraries, and allows developers to separate permissions needed by host apps from those required by the libraries. FLEXDROID identifies the principal of the currently running code using stack inspection and, depending on the identified principal, allows or denies the request by dynamically adjusting the app’s permissions according to the pre-specified permissions in the app’s manifest. However, this approach presents several drawbacks compared to Reaper. The heavy instrumentation of the OS and apps presents a significant obstacle to adoption. Moreover, they require developers to incorporate specific code in the manifest to protect users, but do

not provide developers with incentives to do so. On the other hand, Reaper gives control to the users.

A similar work [34] extended the AppOps manager to provide users with contextual information about the origin of permission requests. Since they do not provide enough technical details we can not compare to their approach. Moreover their library classification is done manually.

TaintDroid [40] and FLOWDROID [25] used dynamic and static taint analysis respectively, for detecting data leaks. PmDroid [43] uses TaintDroid to track and block sensitive data obtained through certain PPCs from being sent to ad networks, but obtains incomplete taint tracking coverage and relies on volatile domain information for identifying ad networks. VetDroid [83] extends the taint tracking logic of TaintDroid to monitor callbacks but suffers from the same coverage limitations. TaintART [73] presented an information flow tracking system integrated inside ART that can be used for detecting data leakage. ARTist [30] is a compiler-based app instrumentation framework that can be used for intra-app taint tracking, as well as dynamic permission enforcement. ArtDroid [37] is a dynamic analysis framework for hooking virtual-method calls supporting both Java and JNI methods. Liu et al. [51] proposed PEDAL, a system that can identify libraries even when the source code is obfuscated. AdDroid [57], Aframe [82], AdSplit [66] and NativeGuard [72] proposed various techniques for separating integrated libraries from the host app. Recon [63] is a VPN-based solution that monitors network traffic to detect and blocks PII exfiltration. MockDroid [31] modified the Android OS so as to replace sensitive information with fake values. Fu et al. [42] proposed a permission policy manager that monitors each library’s method invocation and tracks the execution thread tree. XPrivacy [10] is designed to prevent PII-access but does not distinguish libraries or core functionality; incorporating the origin information produced by Reaper would allow for more fine-grained access control and significantly improve the usability aspect of such tools.

Android Permission Analysis. Wijesekera et al. [80] conducted a user study to understand how often apps require access to protected resources by instrumenting the Android platform. Wang et al. [76] employed text analysis and machine learning to infer how two specific permissions are used based on a manual labelling of 622 apps. They relied on the PScout mappings and reported an accuracy of 85% and 94% for the two permissions. To overcome the obstacle of obfuscated code, they recently incorporated a dynamic analysis aspect and conducted a study on 830 apps [77]. While their approach has similarities with Reaper it presents significant limitations. They rely on a modified version of TaintDroid in Android 4.3 and only perform stack inspection at sink points (e.g., the network). Since stack inspection at this layer does not provide much information about the purpose of the permission due to multithreading, they heavily modified Dalvik to also capture the stacktrace of the parent thread. Their system also induces a slowdown of up to 47% compared to stock Android, while relying on random fuzzy testing which is inherently limited. Reaper has negligible overhead and performs stack inspection at the access level; this allows us to successfully monitor all PPCs for different Android versions, including those based on the ART compiler. Overall, while their study focuses on a different aspect of permissions, incorporating Reaper for their dynamic analysis would allow them to efficiently conduct a large

scale study and achieve higher coverage without the drawbacks of their extensive OS modification.

9 CONCLUSION

Given the ubiquitous presence of smartphones and the massive amount of sensitive information they store, it is imperative to stringently mediate access to user data. Currently, the proliferation and prevalence of third-party libraries renders them a significant privacy risk. To address this issue we developed Reaper, a novel dynamic analysis system that traces the origin of permission-protected calls and non-protected calls that access PII. Our subsequent study on over 5K of the most popular apps, revealed the extent of libraries accessing sensitive data and found that certain permission-protected calls were used exclusively by these libraries and not by the apps’ core functionality. Reaper’s functionality can enhance Android’s fine-grained run time permission system and enable users to prevent third parties from accessing their personal data.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme, under grant agreements, No 740787 (SMESEC), No 786669 (ReAct), No 786890 (THREAT-ARREST) and by the CEF project CertCoop, under grant agreement No. INEA/CE-F/ICT/A2016/1332498. In addition, this project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690972 (PROTASIS). This paper reflects only the view of the authors and the funding bodies are not responsible for any use that may be made of the information it contains.

REFERENCES

- [1] 2011. Xposed Framework API. <https://bit.ly/2L4XmpU>.
- [2] 2012. Over half of 3rd party Android in-app ad libraries have privacy issues and possible security holes. <https://bit.ly/2G3VeJl>.
- [3] 2013. Forbes - Google Users: You’re The Product, Not The Customer. <https://bit.ly/2G7dDM9>.
- [4] 2013. ThreatPost - Unnamed Android Mobile Ad Library Poses Large-Scale Risk. <https://bit.ly/2G5jUrM>.
- [5] 2014. PC World - Researchers: Mobile users at risk from lack of HTTPS use by mobile ad libraries. <https://bit.ly/2BYbzC5>.
- [6] 2014. Xposed Hook Overhead. <https://bit.ly/2BW1HZp>.
- [7] 2016. MobileAppScrutinator: A Simple yet Efficient Dynamic Analysis Approach for Detecting Privacy Leaks across Mobile OSs. <https://bit.ly/2RHTxcC>.
- [8] 2016. A repository of Android libraries. <https://bit.ly/2RCf5Y1>.
- [9] 2016. Root Detection Evasion on iOS and Android. <https://bit.ly/2rpLHZH>.
- [10] 2016. The ultimate privacy manager for Android. <https://bit.ly/2zMdd8a>.
- [11] 2017. The Google Play apps that say they don’t collect your data, and then do. <https://bit.ly/2L0CTCu>.
- [12] 2018. Android Distribution between Platform versions. <https://bit.ly/1kjKifB>.
- [13] 2018. Android library statistics. <https://bit.ly/2L1S5zd>.
- [14] 2018. Android View Class. <https://bit.ly/1ZSHM2L>.
- [15] 2018. AppsFlyer - Mobile App Tracking & Attribution. <https://bit.ly/1lMd3oa>.
- [16] 2018. AppsFlyer SDK Integration - Android. <https://bit.ly/2QjPzKz>.
- [17] 2018. Explorer. <https://bit.ly/2L2XoP3>.
- [18] 2018. GMS, Google’s most popular apps, all in one place. <https://bit.ly/2L6cdk4>.
- [19] 2018. Permission Protection Level. <https://bit.ly/2BWzhif>.
- [20] 2018. Raccoon - APK downloader. <https://bit.ly/1yIT4bR>.
- [21] 2018. UI Automator - Android’s UI testing framework. <https://bit.ly/2B2ze2m>.
- [22] 2018. Window Layout - FLAG_SECURE. <https://bit.ly/2QHEoLk>.
- [23] Jagdish Prasad Acharya, Gergely Acs, and Claude Castelluccia. 2015. On the Unicity of Smartphone Applications. In *WPES ’15*.
- [24] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *ASE ’12*.

- [25] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI '14*.
- [26] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *CCS '12*.
- [27] Tanzirul Azim and Iulian Neamtii. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA '13*.
- [28] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *CCS '16*.
- [29] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Ochteau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *USENIX Security '16*.
- [30] M. Backes, S. Bugiel, O. Schranz, P. v. Styp-Rekowsky, and S. Weisgerber. 2017. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *EuroSP'17*.
- [31] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. Mockdroid: trading privacy for application functionality on smartphones. In *HotMobile '11*.
- [32] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. In *MoST '13*.
- [33] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *FC '16*.
- [34] Saksham Chitkara, Nishad Gothoskar, Suhas Harish, Jason I. Hong, and Yuvraj Agarwal. 2017. Does This App Really Need My Location?: Context-Aware Privacy Management for Smartphones. *IMWUT '17* (2017).
- [35] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *ASE '15*.
- [36] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *NDSS '17*.
- [37] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. In *ESoS '16*.
- [38] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In *NDSS '16*.
- [39] Nicole Eling, Siegfried Rasthofer, Max Kolhagen, Eric Bodden, and Peter Buxmann. 2016. Investigating Users' Reaction to Fine-Grained Data Requests: A Market Experiment. In *HICSS '16*.
- [40] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI '10*.
- [41] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *CCS '11*.
- [42] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang, and Xin Wang. 2017. Perman: Fine-Grained Permission Management for Android Applications. In *ISSRE '17*.
- [43] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. 2015. PmDroid: Permission Supervision for Android Advertising. In *SRDS '15*.
- [44] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *WISEC '12*.
- [45] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *MobiSys '14*.
- [46] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. 2012. A Conundrum of Permissions: Installing Applications on an Android Smartphone. In *FC '12*.
- [47] Ilias Leontiadis, Christos Efstratiou, Marco Picone, and Cecilia Mascolo. 2012. Don't Kill My Ads!: Balancing Privacy in an Ad-supported Mobile Application Market. In *HotMobile '12*.
- [48] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. 2016. Should You Use the App for That?: Comparing the Privacy Implications of App- and Web-based Online Services. In *IMC '16*.
- [49] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *SANER '16*.
- [50] Jiali Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. In *UbiComp '12*.
- [51] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *MobiSys '15*.
- [52] Xing Liu, Sencun Zhu, Wei Wang, and Jiqiang Liu. 2016. Alde: privacy risk analysis of analytics libraries in the android ecosystem. In *SecureComm '16*.
- [53] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *ESEC/FSE '13*.
- [54] Wei Meng, Ren Ding, Simon P. Chung, Steven Han, and Wenke Lee. 2016. The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads. In *NDSS '16*.
- [55] Suman Nath. 2015. MAdScope: Characterizing Mobile In-App Targeted Ads. In *MobiSys '15*.
- [56] Elias P. Papadopoulos, Michalis Diamantaris, Panagiotis Papadopoulos, Thanasis Petsas, Sotiris Ioannidis, and Evangelos P. Markatos. 2017. The Long-Standing Privacy Debate: Mobile Websites vs Mobile Apps. In *WWW '17*.
- [57] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *ASIACCS '12*.
- [58] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *EuroSec '14*.
- [59] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *NDSS '14*.
- [60] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS '16*.
- [61] Vaibhav Rastogi, Yan Chen, and William Enck. 2013. AppPlayground: Automatic Security Analysis of Smartphone Applications. In *CODASPY '13*.
- [62] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan Riley. 2016. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. In *NDSS '16*.
- [63] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *MobiSys '16*.
- [64] Suranga Seneviratne, Harini Kolamunna, and Aruna Seneviratne. 2015. A Measurement Study of Tracking in Paid Mobile Applications. In *WiSec '15*.
- [65] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS '16*.
- [66] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating Smartphone Advertising from Applications. In *USENIX Security '12*.
- [67] Soel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What Mobile Ads Know About Mobile Users. In *NDSS '16*.
- [68] Yihang Song and Urs Hengartner. 2015. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In *SPSM '15*.
- [69] Michael Spreitzenbarth, Felix Freiling, Florian Echter, Thomas Schreck, and Johannes Hoffmann. 2013. Mobile-sandbox: Having a Deeper Look into Android Applications. In *SAC '13*.
- [70] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *MoST '12*.
- [71] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *ESEC/FSE '17*.
- [72] Mengtao Sun and Gang Tan. 2014. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *WiSec '14*.
- [73] Mingshen Sun, Tao Wei, and John Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *CCS '16*.
- [74] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *NDSS '15*.
- [75] Fabo Wang, Yuqing Zhang, Kai Wang, Peng Liu, and Wenjie Wang. 2016. Stay in Your Cage! A Sound Sandbox for Third-Party Libraries on Android. In *ESORICS '16*.
- [76] Haoyu Wang, Jason Hong, and Yao Guo. 2015. Using Text Mining to Infer the Purpose of Permission Use in Mobile Apps. In *UbiComp '15*.
- [77] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. 2017. Understanding the Purpose of Permission Use in Mobile Apps. *TOIS '17* (2017).
- [78] Na Wang, Pamela Wisniewski, Heng Xu, and Jens Grossklags. 2014. Designing the Default Privacy Settings for Facebook Applications. In *CSCW '14*.
- [79] Dominik Wernke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. (2018). <http://arxiv.org/abs/1801.02742>
- [80] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android Permissions Remystified: A Field Study on Contextual Integrity. In *USENIX Security '15*.
- [81] Michelle Y Wong and David Lie. 2016. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS '16*.
- [82] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. Aframe: Isolating advertisements from mobile applications in android. In *ACSAC '13*.
- [83] Yuan Zhang, Min Yang, Bingquan Xu, Zheming Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *CCS '13*.
- [84] Yuri Zhauniarovich and Olga Gadyatskaya. 2016. Small changes, big changes: an updated view on the Android permission system. In *RAID '16*.
- [85] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *SPSM '12*.
- [86] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *CCS '17*.