Elleen Pan*, Jingjing Ren, Martina Lindorfer, Christo Wilson, and David Choffnes

Panoptispy: Characterizing Audio and Video Exfiltration from Android Applications

Abstract: The high-fidelity sensors and ubiquitous internet connectivity offered by mobile devices have facilitated an explosion in mobile apps that rely on multimedia features. However, these sensors can also be used in ways that may violate user's expectations and personal privacy. For example, apps have been caught taking pictures without the user's knowledge and passively listened for inaudible, ultrasonic audio beacons. The developers of mobile device operating systems recognize that sensor data is sensitive, but unfortunately existing permission models only mitigate some of the privacy concerns surrounding multimedia data.

In this work, we present the first large-scale empirical study of media permissions and leaks from Android apps, covering 17,260 apps from Google Play, AppChina, Mi.com, and Anzhi. We study the behavior of these apps using a combination of static and dynamic analysis techniques. Our study reveals several alarming privacy risks in the Android app ecosystem, including apps that over-provision their media permissions and apps that share image and video data with other parties in unexpected ways, without user knowledge or consent. We also identify a previously unreported privacy risk that arises from third-party libraries that record and upload screenshots and videos of the screen without informing the user and without requiring any permissions.

Keywords: privacy; mobile devices; audio, video, and image leaks

DOI Draft manuscript, DOI TBD
Received a date; revised a date; accepted a date.

***Corresponding Author: Elleen Pan:** Northeastern University, E-mail: elleen@ccs.neu.edu
Jingjing Ren: Northeastern University, E-mail: renjj@ccs.neu.edu
Martina Lindorfer: UC Santa Barbara, E-mail: martina@iseclab.org
Christo Wilson: Northeastern University, E-mail: cbw@ccs.neu.edu
David Choffnes: Northeastern University, E-mail: choffnes@ccs.neu.edu

1 Introduction

The high-fidelity sensors and ubiquitous internet connectivity offered by mobile devices have facilitated numerous mobile applications (apps) that rely on multimedia features. For example, a mobile device's camera and microphone enable users to capture and share pictures, videos, and recorded audio. Apps also use these sensors to implement important services such as voice assistants, optical character recognition (OCR), music identification, and face and object recognition.

In addition to such beneficial use cases, apps may use these sensors in ways that violate users' expectations and privacy. For example, some apps take pictures without the user's knowledge by shrinking the viewfinder preview window to a 1×1 pixel, thus making it virtually invisible [51, 68]. Similarly, Silverpush, an advertising company, developed a library that passively listened for inaudible, ultrasonic audio beacons for tracking users' TV viewing habits [28]. Finally, as a possible example of things to come, Facebook has been awarded a patent on using the mobile device's camera to analyze users' emotions while they are browsing the newsfeed [70].

Given that sensor data is highly sensitive, the Android and iOS operating systems include mandatory access control mechanisms around most sensors. However, existing permission models only partially mitigate multimedia privacy concerns because they are *coarse grained* and *incomplete*. For example, when a user grants a multimedia permission to an app, this permission also applies to any third-party library code that is included in the app. Thus, users and even app developers may be unaware of the extent of privacy risks from such permissions. In addition, we find that on Android there is no permission required for third-party code in an app to continuously record the screen displayed to the user. As such, users may unwittingly use apps that collect video recordings containing sensitive information, similar to session-replay scripts on websites [44]. A key challenge for understanding these risks is that there is no general approach to reveal such behavior.

In this work, our goal is to identify and measure the exfiltration of media (defined as images, video, and audio) over the network from Android apps. We focus on

(potential) privacy risks that are caused by the transfer of media recordings to parties over the internet, rather than privacy risks caused solely by apps' access to the camera and microphone (e.g., device fingerprinting [42, 46, 80] and location tracking [28]). We define a leak as either (1) unexpected recording of users' interactions with an app, and (2) sharing of multimedia recordings with other parties over the internet, without explicitly indicating this to the user either in the privacy policy or at run time.

To understand media exfiltration by Android apps and the potential privacy consequences, we empirically studied the behavior of 17,260 apps collected from Google Play and three popular third-party app stores. We analyze these apps using a combination of static and dynamic analysis techniques. We use static analysis on all of the apps in our dataset to determine (1) whether each app requests access to camera and microphone permissions, (2) whether media APIs are actually referenced in the app's code, and (3) whether these API references (if they are present) are in code from the first-party developer or a third-party library. Of course, static analysis alone cannot tell us whether an app actually invokes media APIs, or exfiltrates media over the network. Therefore, we use dynamic analysis (on a subset of 9,100 apps that have the potential to leak media) to detect media exfiltration; specifically, we used the Exerciser Monkey [26] to automatically interact with each app in a controlled environment, recorded network traffic using Mitmproxy [16], and used the MediaExtract file carving tool [6] to identify media in network flows.

Our work makes the following contributions:

- We present the first large-scale empirical study of media permissions and leaks from Android apps, covering 17,260 apps from Google Play, AppChina, Mi.com, and Anzhi.
- We develop a comprehensive methodology for detecting media exfiltration that combines analysis of permissions, method references, third-party libraries, and automated interactions. We validate our methodology by analyzing the behavior of a ground-truth test app that we developed, as well as through manual examination of key apps that are known to rely on image, video, and audio collection.
- We find a previously unreported privacy risk from third-party libraries. Namely, they can record the screen from the app in which they are embedded *without requiring any permissions*. Apps often display sensitive information, so this exposes users to stealthy, undisclosed monitoring by third parties.
- Our analysis reveals that several apps share image and video data with other parties in unexpected ways. For example, several photo editing apps process images in the cloud without explicitly mentioning the behavior in their privacy policy.
- Large fractions of apps request multimedia permissions that they never use, and/or include code that uses multimedia sensors without explicitly requesting permissions for them. This inconsistency increases the potential privacy risks for users: previously unused permissions could be exploited by new third-party code that a developer includes in an app. Further, third-party code that does not have permissions to use multimedia in one version of an app may start exploiting any permissions granted to a future version of the app for an unrelated purpose.

Taken together, our study reveals several alarming privacy risks in the Android app ecosystem. We have responsibly disclosed confirmed privacy leaks to developers and the Android privacy team, and they took action to remediate the privacy concerns we discovered (§7.1).

Our dataset and analysis results are publicly available at `https://recon.meddle.mobi/panoptispy/`.

2 Related Work

We begin by surveying related work on mobile device privacy in general, and media leaks in particular. We also discuss existing approaches and tools for investigating the security and privacy offered by Android apps.

2.1 Privacy Measurements

Tracking and PII collection. Several studies have documented the growing prevalence of tracking in mobile apps. Vallina-Rodriguez et al. presented a broad characterization of the online advertising platforms used by apps [72], and follow-up studies revealed the specific kinds of personally identifiable information (PII) sent to trackers and analytics services [31, 38, 61, 65, 73, 76]. Book et al. investigated APIs exposed by advertising libraries that can be used to leak PII [33]. Ren et al. used longitudinal data to examine how app privacy practices have changed over time [64]. Other studies have focused on legal implications of apps' privacy practices, specifically COPPA and the GDPR [63, 66].

Several studies bridge the gap between tracking on the web and on mobile devices. Leung et al. directly compared the privacy practices of web and app-based versions of the same service [55]. In contrast, two studies have delved into the mechanisms used by advertisers to track users' behavior across devices [34, 81].

While this body of work has significantly advanced our understanding of the mobile tracking ecosystem, one shortcoming is that it exclusively focuses on leaks of textual information to third parties (e.g., unique identifiers, email addresses, names, etc.).

Attacks using multimedia sensors. Several previous studies take an initial look at how a mobile device's cameras and microphones can be used to violate user privacy and security. For example, unintentional variations in the manufacturing of mobile device cameras, microphones, and speakers can be used to create fingerprints that uniquely identify mobile devices [42, 46, 80]. Petracca et al. demonstrated numerous attacks that apps with microphone permissions can implement by passively eavesdropping in the background [60]. Similarly, Fiebig et al. demonstrated that apps with camera permissions could passively capture keystrokes and even users' fingerprints [45].

Two studies have examined the deployment and implications of ultrasonic beacons. Arp et al. measured the prevalence of ultrasonic beacons in the wild, and found them deployed on websites and in stores. Furthermore, they found 234 apps in the Google Play Store that were constantly, passively monitoring for these beacons, in order to track users' online and offline browsing behaviors [28]. Mavroudis et al. consider various attacks against users that leverage ultrasonic beacons, including de-anonymizing Tor users [59].

Shrivastava et al. developed a testing framework that probes the computer vision algorithms used by apps with camera permissions [67]. They found that many apps included libraries that implement character, face, and barcode detection. Furthermore, the authors surveyed users and found that 19% of apps in their study extracted information from images that users did not expect, and that this made users very uncomfortable.

Our work. Our study complements and extends the existing measurement work on the privacy implications of media sensors on mobile devices in two significant ways. First, existing studies focus on how apps can extract and distill privacy-sensitive data from images and audio (e.g., fingerprints). In contrast, we focus on the wholesale exfiltration of media over the internet. Second, prior work does not consider the privacy implications of static screenshots and captured videos of the screen. As we will show, these represent significant privacy risks since they can be surreptitiously recorded by any app without the need for explicit permissions.

2.2 Privacy Measurement and Tools

Numerous tools from the research community help identify, and in some cases mitigate, security and privacy risks on mobile devices.

Static analysis. Previous work analyzed the privacy implications of Android app bytecode using a variety of static analysis techniques, such as static data flow (taint) analysis [29, 36, 47, 52], and symbolic execution [50, 78]. These systems uncover many PII leaks, but they often overestimate the number of leaks, thus leading to false positives. Further, code that is heavily obfuscated or dynamically loaded at run time can lead to false negatives (recent measurements indicate that 30% of Android apps load code at run time [56]).

Dynamic taint analysis. TaintDroid was the first system to pioneer the use of dynamic taint tracking to analyze privacy leaks on Android [43]. Subsequent systems have refined these dynamic analysis techniques [75, 77, 79]. Additionally, there are several tools to assist in automating the testing process for Android apps, i.e., to increase code coverage when performing taint analysis [37, 39, 48, 49, 58]. Unfortunately, dynamic analysis alone suffers from false negatives, as fully exercising all code paths in complex apps is generally not feasible. Further, taint tracking imposes run-time overheads that make it challenging to run analysis at large scale in a reasonable amount of time.

Dynamic network traffic analysis. A separate line of work focuses on identifying privacy leaks in network traffic [54, 63, 65, 69]. The advantage is that these approaches are easily deployable for end-user devices, either via a Virtual Private Network (VPN) proxy or by conducting analysis on a home router. When combined with ground-truth information about PII and/or machine learning, this approach can provide good coverage of privacy leaks with few false positives and negatives. However, such approaches will not work well if the PII is exfiltrated using sophisticated obfuscation [40].

Our work. No single method is totally effective at detecting all privacy leaks from Android apps. Thus, in this study we leverage a combination of static analysis and dynamic network traffic analysis to measure media leaks. As we discuss in § 5, we first use static analysis

to examine the permissions requested by apps and references to sensitive API calls. We then run the apps and automatically interact with them in an attempt to trigger those APIs, and subsequently analyze the corresponding network traffic that those apps generate to identify media leaks.

3 Threat Model

Our goal is to identify and measure exfiltration of media (i.e., images, audio, and video) by Android apps over the network. Media exfiltration presents new privacy implications compared to well-known PII leaks. They provide an extra channel to carry PII and private information (e.g., a user's images) that prior approaches do not identify. Furthermore, screen recording reveals data as it is entered, which the user may reasonably expect not to be shared until submitted. Finally, screen recording might reveal highly sensitive information, such as passwords: Android has the option to toggle password visibility globally in its security settings (i.e., showing the entered characters briefly before masking them) or locally for individual input fields (i.e., unmasking the whole password) if enabled by the developer.

Definition of media leaks. We assume that the user has either granted no permissions, or granted an app permissions to use media sensor(s) for a user-identifiable purpose of that app. For example, a user would grant no media permissions to a simple Solitaire app, and would grant camera permissions to an app that allows the user to take and edit photos. A *suspicious* or *unexpected* media exfiltration is one that meets at least one of the following criteria:

- *It does not further the primary purpose of the app.* Media shared outside of an app's primary purpose presents privacy risks since users do not expect it. In many cases, this shairng is due to third-party tracking or analytics libraries. For other cases, we manually inspect the app being studied to assess this property.
- *It is not disclosed to the user.* Media sharing that is not disclosed may not only be unexpected by the user, but also may violate privacy laws. We manually verify whether an app provides visual cues to users, requests users' consent, and/or clearly discloses this behavior in its privacy policy.
- *It is not employed by similar apps.* We determine this based on comparisons with apps that have nearly identical functionality. If other, similar apps do not exfiltrate media, then it is a good indicator that such functionality is suspicious. We then manually investigate and subjectively label such cases.
- *It is not encrypted over the internet.* This creates opportunities for eavesdroppers to passively observe sensitive content. We check this property based on whether media is sent over an unencrypted channel.

We assume that apps do not attempt to break the permission model, nor break out of the Android sandbox (e.g., by exploiting OS-level vulnerabilities). We further assume that apps access media sensors using only standard Android APIs that are available to all app developers on recent Android platforms, as opposed to hidden or privileged APIs. We do not examine media exfiltration from apps' background activity. We also do not consider data that is reshared after collection, as was the case for the Cambridge Analytica controversy.

Privacy legislation. While we do not provide a legal analysis of privacy leaks in this study, our definition of leaks is in line with recent legislation that requires companies to disclose and explain the purpose of collected PII. The European Union's General Data Protection Regulation (GDPR) restricts and requires full disclosure of PII collection and usage [11]. The California Online Privacy Protection Act (CalOPPA) requires any party who collects PII from Californian consumers to provide a privacy policy outlining what data is collected and who it is shared with, and to comply with posted policies [5]. The Fair Information Practice Principles is a set of principles adopted by the US Privacy Act and other frameworks worldwide. It details principles such as transparency, purpose specification, and data minimization, among others [8].

4 Background

Before we describe our methodology for investigating media leaks from Android apps, it is important to review the permission model and APIs offered by Android to access media resources.

Media permissions. Android restricts access to sensitive OS capabilities by forcing developers to obtain explicit permission from users. App developers must list the permissions they plan to use in their

AndroidManifest.xml file, which is contained in all Android Packages (APKs). To access the camera and microphone, apps must request the following permissions:

- android.permission.CAMERA
- android.permission.RECORD_AUDIO

Additionally, apps may request the permissions android.permission.READ_EXTERNAL_STORAGE or android.permission.WRITE_EXTERNAL_STORAGE to access files that are stored on the device. This poses another possible outlet for media leaks, as apps can access and potentially leak photos, videos, or audio clips stored on the device if granted either of these permissions. Note that in the Android permission model, the permission to write to external storage implicitly grants read access.

Users can accept or reject permission requests. Prior to Android API level 23, permission requests needed approval at app install time, and rejection prevented installation. Since API level 23, apps request permissions (and must handle rejection) at run time.

Media APIs. Once an app has been granted media permissions, the following API objects become available:

- android.hardware.camera (API level <21)
- android.hardware.camera2 (API level 21+)
- android.media.AudioRecord
- android.media.MediaRecorder

The camera and AudioRecord objects require the CAMERA and RECORD_AUDIO permissions, respectively. The MediaRecorder object only requires RECORD_AUDIO if used solely for audio recording. Otherwise, to record video, both permissions are required.

Screenshots. Unlike the camera and audio APIs, the APIs for taking screenshots and recording video of the screen *are not protected by any permission*. The Android APIs for capturing the screen are:

- android.view.View.getDrawingCache()
- android.view.View.getRootView()

This lack of access control is problematic, as apps can potentially record users' screen interactions without their awareness. However, these two methods are multipurpose and not solely designed for taking screenshots. For example, getDrawingCache() caches a bitmap, which is useful for improving performance when rendering repeated UI elements between activities. The method getRootView() finds the topmost view of the UI's layout, which is a hierarchical tree structure consisting of ViewGroups (internal nodes) and Views (leaf nodes). In short, when an app calls these methods it does not necessarily imply that it is recording the screen.

Note that this approach of capturing the screen is different from that of Android's MediaProjection API. The latter provides means to record the screen programmatically, but includes an indication in the form of a lock icon. Since the user is informed about the recording in this case, this API is outside of our threat model.

5 Methodology

In this section, we present our methodology for gathering data and measuring media leaks by Android apps. As shown in Figure 1, our methodology involves both static and dynamic analysis techniques. We begin by describing our process for gathering Android apps for analysis in § 5.1. Next, we discuss our approach for extracting permissions and method usage from APKs using static analysis in § 5.2, and our dynamic testbed for automatically interacting with apps and inducing media exfiltration over the network in § 5.3. Finally, in § 5.4 we explain and validate our approach for detecting media in network flows.

5.1 Selection of Android Apps

Obtaining a broad understanding of media leaks requires testing a large set of apps. However, the time and resources necessary to dynamically analyze apps is non-trivial (see § 5.3), and thus we must carefully choose how to allocate our limited resources.

To provide analytical results that are representative of apps in general, while also covering high-impact apps, we select popular and random apps from four app stores. Our set of apps is compiled from several preexisting sources [27, 40, 64], and covers apps from Google Play, AppChina, Mi.com, and Anzhi. We chose these three third-party app stores because they were the three largest markets (aside from the Google Play) in the AndroZoo dataset [27].

From Google Play, we select 8,038 apps that request permissions for the camera and/or microphone from a set of 30,504 apps that are either part of the top 600 popular free apps, top 600 popular apps for each category, newest 600 overall, or newest 600 in each category as of April 2017 [40]. We further include 7,665 APKs collected from a previous study [64] that were either part of the top 600 popular free apps or the top 50 in each cat-

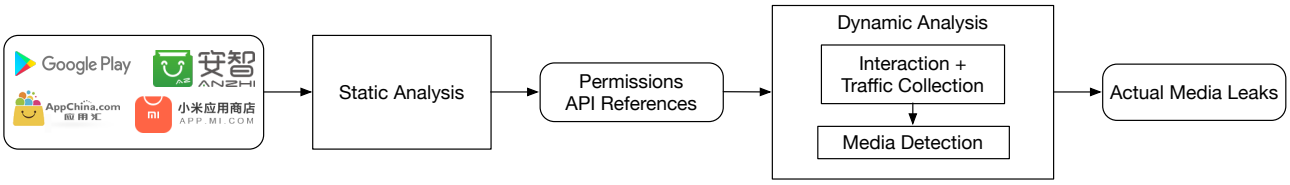

Fig. 1. Design of our experiments. We start with 17,260 apps collected from four app stores on the left. We statically analyze these apps to extract the media permissions and API references, which then informs our selection of apps to dynamically analyze. The final output, on the right, are the actual media leaks from apps over the network.

egory as of January 2017. The final Google Play dataset covers 15,627 unique APKs. From the third-party app stores, we select the most popular apps as well as 1,000 apps selected uniformly at random from AndroZoo [27]. Specifically, we collect the 510 most popular apps overall from AppChina, and the most popular apps from each category from Mi.com (528 apps) and Anzhi (285 apps). In total our dataset contains 17,260 unique apps.

5.2 Static Analysis

The next step is statically analyzing the 17,260 apps in our dataset. We use static analysis to determine:

1. Does the app request permissions for the camera, microphone, and/or accessing external storage?
2. Does its bytecode contain references to the media APIs listed in § 4?
3. Are media API references in third-party library code, and if so, which library?

We now discuss why each of these pieces of information is important for our analysis, and how we obtain them.

Permissions. Examining permissions is the first step towards understanding which apps in our dataset *might* leak images, audio, and video, since permissions are required to access these sensors or files stored in the external storage. We use the standard Android SDK tool `aapt` to retrieve the `AndroidManifest.xml` file from all of the apps in our dataset, and scan the results for apps that request permissions to access the camera, microphone, or external storage.

However, an app that requests such permissions does not necessarily use the corresponding media APIs or leak media over the network. This can occur when apps request permissions for functionality that is never used by the app, i.e., the apps are over-privileged [35]. Further, apps that do not request these permissions may still potentially leak media, e.g., if they upload images from the mobile device's internal storage, or gather and upload screen captures. As a result, our static analysis on permissions may have false positives and negatives, which we control for with later dynamic analysis.

API references. We decompile the apps in our dataset using dex-method-list [6] and locate references to the camera, audio, and screen capture APIs listed in § 4. This allows us to identify apps that are over-privileged, as well as apps that may be capturing screenshots and screen video. However, the methods for capturing/recording the screen and reading data from device storage may serve other purposes, meaning that the static analysis produces a high false positive rate for API references to screenshot functionality and reading from external storage. As a result, we also perform dynamic analysis on these apps, described in § 5.3.

Third-party libraries. Android apps often include third-party libraries, some of which have been shown to be the root causes of privacy leaks (e.g., advertising and tracking libraries [32]). Libraries are able to access sensitive information on mobile devices because they inherit the capabilities of the app itself. This raises the possibility that library code may take actions that users, and even first-party developers, are unaware of.

In the context of this study, we are interested in whether references to media APIs are within code from the first-party app developer or a third-party library. This information is critical for correctly attributing the source of media leaks. To identify the libraries within apps, we rely on LibScout [30] and LibRadar [57]. Both tools compare the signatures of bytecode against a predefined database of known library code. Unfortunately, because of bytecode obfuscation and the presence of previously unknown library versions, both tools may produce false negatives. Furthermore, these tools may produce false positives if an app includes a library, but never invokes its methods at run time.

To determine whether media API references occur in first or third-party code, we rely on package names. Typically, code from the first-party developer resides in a package name that largely overlaps with

Dataset	App Source	# of APKs	Selection Criteria
method-call	Google Play	127	Apps that call camera and audio APIs
3p-lib	Google Play	187	Apps that covered the most popular set of third-party libraries
appsee	Google Play	33	Apps that include the AppSee library
permission	Google Play	8,038	Apps that request either camera and audio permission
appchina	AppChina	335	Apps that request either camera or audio permission, or call screenshot methods
appmi	Mi.com	331	Apps that request either camera or audio permission, or call screenshot methods
anzhi	Anzhi	269	Apps that request either camera or audio permission, or call screenshot methods

Table 1. Summary of the 9,100 apps we selected for dynamic analysis, and the criteria used for their selection. Some of our datasets (3p-lib and appsee) overlap with the rest of our dataset as we selected them for further testing after initial results.

the application package name. We rely on this assumption to distinguish code from first- and third-parties. For example, all classes related to the main activity of the app `air.com.myheritage.mobile` are under the same package name, yet it also includes packages corresponding to third-party libraries like `com.appsee` and `com.google.android.gms.maps`.

Privacy policies. Our definition of media leaks relies on app privacy policies (§ 3), so we manually inspect the privacy policies of apps that share media over the internet. If this type of sharing is not explicitly disclosed in the app's privacy policy, we call it a media leak.

5.3 Testbed for Dynamic Analysis

Static analysis provides useful guidance about which apps may potentially exfiltrate media. However, from this data alone we cannot infer whether media permissions will be used, or whether media APIs will be called at run time. Thus, results from static analysis alone may exhibit high false positive rates. On the other hand, static analysis fails to detect obfuscated and dynamically loaded code, causing false negatives. To address these issues, we conduct dynamic analysis by running and interacting with apps. Due to resource constraints,[1] we are not able to dynamically analyze all 17,260 apps; instead, as shown in Figure 1, we select apps that are more likely to leak media content based on their permissions and media API references. We dynamically analyzed 9,100 apps (53% of our total dataset). Table 1 shows how these apps are distributed across app store sources, as well as the criteria for their selection.

In the remainder of this section, we describe our testbed for dynamically analyzing Android apps.

Automated interaction. Triggering media exfiltration from mobile apps requires executing and interacting with them. A natural way to accomplish this is via human interaction; however, this does not scale to the size of our dataset. Instead, we use the UI/Application Exerciser Monkey [26]. Each test consists of interacting with an app using Monkey for 5,000 user events (lasting for 16 minutes at most). We configured Monkey to randomly select 10 activities in each app and send 500 interactions to each activity. We use 5,000 events because it allows us to test a large number of APKs in a reasonable amount of time, and because previous work found that longer interaction times did not result in more PII leaks [55]. Note that we did not use pre-configured text inputs, which vary across apps and require substantial manual effort; instead, we relied on random interactions. Accordingly, we miss some events that only human interactions trigger, e.g., in apps that require login.

During each test, we took screenshots from each device at 1-second intervals. We use these screenshots to manually verify that observed media exfiltration was not caused by an explicit interaction event (e.g., clicking the "upload image" button in an app).

Test environment. We conduct experiments using ten Android devices: two Nexus 6P phones and six Nexus 5X phones with Android 6 (API level 23), and two Nexus 5 phones with Android 4.4.4 (API level 19). We use real Android devices instead of emulators to avoid scenarios where apps and third-party libraries behave differently when they detect emulation. We randomly assigned apps to devices; 1,814 were ultimately tested under Android 4.4.4.

Each test was performed in a standardized environment. Before each test, we prepared the device by deleting all non-standard apps (i.e., everything except for the standard app suite provided by Google), clearing the internal user-accessible storage, and then preloading several media files (two decoy Grace Hopper images, a short video clip, and a short audio clip). These me-

1 We conduct all dynamic tests on actual Android devices, and each test takes on the order of minutes.

Category	Supported	Unsupported
Audio	**3gp**, **aac**, id3v2, **m4a**, **ogg**, wav	**raw**
Image	**bmp**, gif, **jpg**, **png**, **webp**	
Video	**3gp**, **mp4**, **webm**	

Table 2. Media file types supported by our augmented version of MediaExtract, based on encoders supported by the Android APIs (bolded) and common libraries we observe in practice.

dia files were placed in the standard locations within the Android filesystem (e.g., `/sdcard/Pictures`). We preloaded the test devices with media as a means to catch apps that exfiltrate media from the filesystem without recording any media themselves. Once the device is cleaned and preloaded, we installed the target app and exercised it with Monkey.

Recording network traffic. During each test, we route network traffic through Meddle [62] using a VPN, and use Mitmproxy [16] to record the plaintext content of HTTP and HTTPS flows. For apps that prevent TLS interception via certificate pinning, we use JustTrustMe [13], which modifies Android to bypass certificate pinning for apps that leverage built-in Android networking APIs and popular libraries (e.g., OkHttp).

5.4 Detection of Media in Network Traffic

Our dynamic tests produce a large dataset of plaintext network flows generated by apps. In this section, we discuss how we identified media embedded in these flows.

5.4.1 Media File Extraction and Decoding

We retrieved the raw byte streams of payload content from each outgoing network flow (typically the payloads of HTTP POST and PUT messages). We then scanned these byte streams with MediaExtract [15] to extract embedded media files. MediaExtract identifies media files by looking for the "magic numbers" that signify the beginning of media file headers. For example, JPEG files are always prefaced with the hexadecimal bytes "FF D8 FF". We modified MediaExtract to support two additional file types: WebP and WebM. We also evaluated several other forensics tools (Autopsy [4], TestDisk/PhotoRec [18], Foremost [9], Scalpel [23], tcpxtract [24], LaZy_NT [14], PIL [20]), but these tools either supported fewer file formats than MediaExtract, identified fewer media files in our data than MediaExtract, or extracted incomplete and corrupted media files.

Table 2 shows the media file types that can be natively produced by the Android APIs, as well as the file types supported by our augmented version of MediaExtract. We are able to detect all file formats that Android can natively produce, except for raw audio because it does not have a distinguishable file header. Fortunately, it is unlikely that apps will attempt to upload raw audio over the network because it is uncompressed, and the file sizes are large compared to other audio formats.

As with all file carving tools, MediaExtract may produce false positives, i.e., files that it incorrectly labels as media. We verified that all extracted image files were true positives by manually checking the media content, e.g., by opening an extracted image file. We then repeated experiments manually to ensure observed leaks were repeatable. Further, we manually determined that all extracted audio files $\leq$1KB in size were false positives. We did not find any true positive audio files in our extracted dataset, i.e., no apps appeared to exfiltrate audio in our tests. We also verified the origin and destination of the network flow carrying the media files to ensure that the traffic comes from the app itself, as opposed to a background service or a stock app.

Other encodings. We noticed that some flows in our dataset relied on specialized encoding formats. We manually verified that MediaExtract was able to locate media embedded in Protocol Buffer [22] and Thrift [1] RPC data structures. Similarly, we pre-processed flows to decode base64-encoded data before running MediaExtract.

5.4.2 Validation

We use controlled tests and manual experiments to validate our extraction of media files from network flows.

Test app. We wrote a simple Android app that could produce all supported types of images, video, and audio files (see Table 2) and upload them to a web server. We ran this app through our data collection infrastructure (i.e., Meddle and Mitmproxy) and attempted to recover the files with MediaExtract. With the exception of raw audio, we were able to recover all of the uploaded files.

Manual tests. We generated network traces with well-known apps that we knew would upload media, such as Imgur and Giphy (images), SoundCloud (audio), and Sing! by Smule (audio & video). We were able to recover all images and videos, as well as audio files that were uploaded in full. However, there were cases where we could not recover audio data. For example, Shazam

Store	# of Apps	Audio		Camera		Screen Capture APIs		External Storage Access
		Permission	API	Permission	API	Screenshot	Video	Permission
Anzhi	883	12.8%	9.7%	15.7%	11.7%	20.7%	1.5%	23.4%
AppChina	468	28.4%	22.9%	37.0%	28.6%	57.1%	2.4%	94.0%
Mi.com	392	55.9%	41.8%	61.0%	45.7%	81.6%	5.6%	97.4%
Google Play	15,627	**45.7%**	**46.2%**	80.5%	75.1%	89.1%	10.6%	92.7%
All	17,260	43.8%	43.6%	75.6%	70.1%	84.6%	9.8%	89.9%

Table 3. Media permission requests and media API references for the app stores in our study. Large fractions of apps request permissions for media; in general, a smaller fraction actually call methods that use those permissions. A notable exception is the audio permission—many apps include code that calls audio APIs but do not request permissions for it (bold text in the table).

interspersed small chunks that appear to be an audio signature, alongside metadata in JSON structures. Interestingly, voice assistants like Hound and Robin did not upload audio at all; instead they transcribed it locally on the mobile device and uploaded the text.

6 Aggregate Results

In this section, we present aggregate statistics for our analysis of media leaks. We begin by investigating the correlation between media permissions requested and code references to media-related APIs (§ 6.1), then analyze which libraries call these APIs (§ 6.2). Last, we use dynamic analysis to determine the media leaks detected in network traffic (§ 6.3).

6.1 Permissions and API References

Our first step in understanding the potential for media leaks is to analyze which media permissions each app requests, and which media APIs appear in the app's code. We summarize the fraction of apps that request audio and camera permissions, and that call methods to capture media, in Table 3. Each row corresponds to a different app store, and the Audio and Camera columns specify the fraction of apps in each store that requests a corresponding permission and that calls a corresponding API. The Screen Capture APIs columns refer to methods that are used for taking a screenshot or recording a screen video, neither of which require permissions. The rightmost column lists the fraction of apps that request read or write permission for external storage.

The last row aggregates results over all apps in our study. We find that among the popular and randomly selected apps, a significant fraction of apps requests media permissions (43.8% for audio and 75.6% for camera). However, this is biased towards apps from Google Play. Among the Chinese app stores, apps from Mi.com have similar permissions requests compared to apps from Google Play; for the other two stores, the rates of permission requests are much lower.

A notable trend is that larger fractions of apps request media permissions than actually call media APIs (on average), which means apps may declare the permissions but never actually use them. Such practices could impose additional risks, since third-party libraries can potentially load dynamic code to abuse the granted permissions without developers or users knowing.

Note that method references do not necessarily mean that the method is called. Likewise, a third-party library may be included, but never used. We speculate that such practices explain the higher percentage of method references than permission requests for audio resources (bold text in Table 3).

Furthermore, APIs for taking screenshots and reading from device storage also serve other purposes, which produces a high false positive rate. For example, methods for reading from device storage are called in 96.1% of our app set, i.e., 16,580 apps call either `getExternalStorage` or `MediaStore`.

To summarize, significant fractions of apps request media permissions and include code that can use them. Interestingly, there is a nontrivial amount of inconsistency between permissions and API calls, and thus a need for developers to more carefully consider how they request and use media functionality. We speculate the reasons for over-provisioned permissions may come from several sources. For one, an app may have required the permission only in a previous version, but developers failed to update requested permissions in the current version. Also, the mapping between Android permissions and their associated API is surprisingly poorly documented, potentially leading to developer confusion. Last, third-party SDKs provide copy-and-paste instructions for integration that includes all potentially needed permissions even if the developer does not use library functionality that requires them.

Library	# of Apps	% of Apps	% of Apps Referencing API from the Library			
			Audio	Camera	Screenshot	Video
com.facebook	8,322	48.22%	0.04%	0.64%	4.54%	0.37%
com.google.android.gms.maps	7,825	45.34%	0	0	0.01%	0
rx	3,602	20.87%	0	0.03%	0.06%	0
com.inmobi	2,411	13.97%	17.13%	0	26.59%	0
com.google.android.gms.vision	1387	8.04%	0	87.60%	0	0
com.tencent.mm	1,316	7.62%	0	0	0.08%	0
com.millennialmedia	1,272	7.37%	0	0	31.29%	0
com.mopub	1,175	6.81%	0	0	45.87%	0
uk.co.senab.photoview	1,163	6.74%	0	0	0.77%	0
net.hockeyapp.android	967	5.60%	0	0	59.77%	0
com.mixpanel.android	853	4.94%	0	0	71.51%	0
com.tapjoy	621	3.60%	0	0	58.13%	0
com.amazon.device.ads	396	2.29%	0	0	62.12%	0
com.smaato.soma	237	1.37%	0	0	97.47%	0
cn.domob	123	0.71%	0	0	86.18%	0
com.adsdk.sdk	105	0.61%	0	0	92.38%	0
com.mdotm.android	58	0.34%	0	0	27.59%	0
com.heyzap	51	0.30%	0	0	19.61%	0
com.mapbox.mapboxsdk	39	0.23%	0	0	12.82%	0
com.skplanet.tad	31	0.18%	0	0	87.10%	0
com.fusepowered	11	0.06%	9.09%	0	100.00%	0
com.tapit	10	0.06%	0	0	100.00%	0
com.noqoush.adfalcon.android.sdk	5	0.03%	0	0	60.00%	0
com.appflood	3	0.02%	0	0	33.33%	0
com.vdopia.ads.lw	3	0.02%	0	0	100.00%	0

Table 4. Identified third-party libraries in our dataset, and the fraction of apps whose library version references media APIs. Of the 163 libraries identified, only the above 25 reference media APIs. Libraries exhibit a diverse set of media API requests across apps, likely due to different versions of libraries and developer customization.

6.2 Third-party Libraries

It is common practice for apps to include third-party libraries for purposes such as utility functions, analytics, and advertising. In many cases, developers may have a limited (or no) understanding of the code contained in these libraries. As such, third-party libraries can be an interesting vector for media leaks.

We investigated the risks from third-party libraries by analyzing their code for references to media APIs. Using LibScout, we identified 163 unique libraries based on their signatures from 17,260 apps. We then matched these libraries with path names identified by dexmethod-list on the files. Note that our list of libraries is incomplete because both library package names and library method calls might be obfuscated at compile time, preventing us from properly identifying the library. This is a challenging and orthogonal research problem [74]. Furthermore, LibScout can only identify libraries in its signature database, which does not include the libraries we discuss in detail in §7. For the libraries we could automatically identify, we focus on any references in the library path to media APIs. Table 4 shows the percentage of apps that include third-party libraries and those that call media API(s) in the third-party library path. We omitted Android libraries and third-party libraries that do not use media APIs (138/163) from the table, which account for the majority of libraries.

Among the 25 libraries, we observe a diverse set of behaviors for permission requests and API calls. Only `com.facebook` includes references to every category of media API. Few libraries include code that accesses the microphone: `com.facebook`, `com.google.android.gms.maps`, and `com.tencent.mm`. Only `com.facebook`, `rx`, and `com.google.android.gms.vision` reference camera APIs, while only `com.facebook` references video APIs. Note that the video API (`MediaRecorder`) may also be used for audio recording. Almost all of the libraries reference the APIs that can be used to capture screenshots; however, we caution that these APIs have other uses besides recording the screen.

Notably, references to media APIs for the same third-party library can differ widely depending on which app included the library. We believe this may be due to different versions of libraries providing different functionality, or developers who customize the code included in their apps.

App	Domain	Request Method	Media Type	Description
kr.kkh.image_search2	images.google.com	POST (HTTPS)	png	expected, image search
com.mnnapps.twinfinder_lookalike	www.google.com	POST (HTTPS)	jpg	expected, image search
com.allintheloop.sahic	collector-10.testfairy.com	POST (HTTPS)	jpg	**unexpected, screenshot image of app usage**
com.smaper.artisto	artisto.smaper.com	POST (HTTPS)	jpg	**unexpected, photo editing**
com.fotoable.fotobeauty	paintlab.fotoable.net	POST **(HTTP)**	jpg	**unexpected, photo editing**
com.allintheloop.sahic	collector-7.testfairy.com	POST (HTTPS)	jpg	**unexpected, screenshot image of app usage**
innmov.babymanager	babymanagerapp.com	POST (HTTPS)	jpg	expected, sharing screenshot
com.umonistudio.tile	log.umsns.com	POST **(HTTP)**	jpg	expected, sharing screenshot of game score
com.facebook.moments	api.facebook.com	POST (HTTPS)	jpg	expected, photo upload
com.kodakalaris.kodakmomentsapp	kodakmoments.kodakalaris.com	POST (HTTPS)	jpg	expected, photo upload
com.goodreads	match-visualsearch.amazon.com	POST (HTTPS)	jpg	expected, image search
com.main.gopuff	c6e83853...0b.api.appsee.com	POST (HTTPS)	mp4	**unexpected, screenshot video of app usage**
com.picas.photo.artfilter.android	api.picas.tech	POST **(HTTP)**	jpg	**unexpected, photo editing**
io.faceapp	node-03.faceapp.io	POST (HTTPS)	jpg	**unexpected, photo editing**
com.neuralprisma	api2.neuralprisma.com	POST (HTTPS)	jpg	**unexpected, photo editing**
io.anyline.examples.store	anyline-tracking.azurewebsites.net	POST (HTTPS)	jpg	expected, photo-to-text scanner
com.hound.android.app	bh.houndify.com	POST (HTTPS)	jpg	**unexpected, screenshot image of app usage**
com.msearcher.camfind	api.camfindapp.com	POST **(HTTP)**	jpg	expected, image search
com.momento.cam	selfy.s3.amazonaws.com	PUT (HTTPS)	jpg	expected, photo upload
com.intsig.BizCardReader	vcf.intsig.net	POST (HTTPS)	jpg	expected, business card scanner
com.zazzle	up.zazzle.com	POST **(HTTP)**	jpg	expected, photo upload

Table 5. Summary of detected media in app-generated network traffic. Of the 21 cases, we find 12 to be leaks (bolded in the first column): they are either unexpected media transmissions (noted in the last column) or sent in plaintext (bolded in the "Request Method" column), exposing potentially sensitive information to eavesdroppers.

6.3 Media in Network Traffic

Next, we analyze the network traffic generated by the 9,100 apps that we analyzed dynamically (as described in §5.3). Table 1 summarizes the apps we selected for dynamic analysis and the criteria we used to do so.

Recall that our testbed gathers all the network traffic generated during automatic interactions with these apps, and we search network flows for media content. Table 5 shows the list of apps (identified by package name in the first column) that transmitted media content during our tests. The second column specifies the destination domain that received the media content, followed by the HTTP method and whether encryption was used. The fourth column specifies what type of media was transmitted, and the last column indicates our analysis of whether the transmission was intentional (and thus expected) or not, and what kind of media sharing was identified.

We use bold text in the last column to highlight nine cases that leak media. These include uploading photos, screenshots, or even videos of screen interactions. The bold rows in the third column highlight additional five cases in which the media content is sent in plaintext, meaning a network eavesdropper (e.g., on a public WiFi access point or in the user's ISP) can also see the media that is transmitted.

Of the 21 cases of media leaks, just under half (9) are shared with third parties that the user may not be aware of. Among the third-party domains, we observe third-party libraries and cloud services (AWS and Azure).

6.4 Analysis of Large Network Flows

The previous analysis relied on identifying known media types in network traffic, but could miss cases where the media encoding is non-standard, obfuscated, or encrypted at the application layer. In this case, an alternative approach to detect potential media content is to look at relatively large flows that could correspond to images, audio recordings, or videos.

We begin by plotting the size of each network flow generated during dynamic analysis. We remove flows generated by Google Play Services from this analysis. Although these flows are large and frequent, we do not consider them to be a vector for media leaks. Figure 2 shows the resulting CDF of the number of bytes per flow across all apps. The vast majority (99.81%) of requests are no larger than 100 KB and more than 80% contain fewer than 10 KB. By comparison, the size of extracted images in our study ranges from 8.2 KB to 1.1 MB.

We further investigated the content of the relatively large flows (≥100 KB) in our dataset, which are sent to 16 second-level domains (7 of which are third-party domains), and 12 of which have more than one large flow (see Table 6). Table 7 lists the apps responsible for those flows. A notable case is the domain `skyhookwireless.com` that is contacted by multiple apps and provides services to locate devices (e.g., IoT devices). The content of the large HTTP requests is an XML file with information about nearby access points (MAC, SSID, signal strength and age) that can be used to calculate fine-grained geolocations without

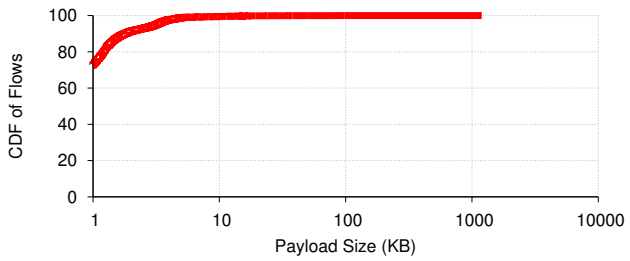

Fig. 2. CDF of payload size per flow for data sent from the app to the internet. The vast majority of flows are small (as expected), but the minority of large flows indicates potentially significant data exfiltration.

Domain	Average Size (KB)	# of Flows
radarstick.com	1,190	4
camfindapp.com	1,070	2
kodakalaris.com	1,069	2
*hockeyapp.net	428	1
faceapp.io	308	2
*skyhookwireless.com	289	7
midomi.com	224	5
mysoluto.com	200	24
*google.com	170	52
houndify.com	158	1
*crittercism.com	131	2
smaper.com	118	3
*newrelic.com	110	1
*googleapis.com	102	1
*appsee.com	101	28
marcopolo.me	101	10

Table 6. Second-level domains receiving large requests of at least 100 KB. (*) indicates the domain belongs to a third party.

needing to access GPS. Manual investigation of other large flows revealed that they generally contained detailed information about the device and apps, at a level that third-party domains can use to fingerprint users [41, 53, 71]. While outside the scope of our study of media leaks, these large flows represent an additional privacy risk that users should be aware of. Further, such large flows can potentially use substantial portions of a cellular data plan's quota.

Crucially, manual analysis of all of these large flows did not reveal any additional exfiltrated media files. This is a positive sign, which suggests that the false negative rate of our media-detection methodology is low.

7 Case Studies

The previous section focused on aggregate information about media leaks that we observed in our dataset. In this section, we use case studies to highlight several interesting media leaks in detail, identify their root causes, and discuss their privacy implications.

Domain	Package Name of App
appsee.com	com.main.gopuff
camfindapp.com	com.msearcher.camfind
crittercism.com	com.usaa.mobile.android.usaa
faceapp.io	io.faceapp
google.com	kr.kkh.image_search2
google.com	com.mnnapps.twinfinder_lookalike
google.com	com.midoapps.cartooneditor
google.com	meemtech.flashlight
googleapis.com	com.eosmobi.cleaner
hockeyapp.net	org.becu.androidapp
houndify.com	com.hound.android.app
kodakalaris.com	com.kodakalaris.kodakmomentsapp
marcopolo.me	co.happybits.marcopolo
midomi.com	com.melodis.midomiMusicIdentifier.freemium
mysoluto.com	com.asurion.solutohome.walmart
mysoluto.com	com.asurion.solutohome.gigspartner
newrelic.com	com.traegergrills.app
radarstick.com	com.radarworkx.radarspotter
skyhookwireless.com	air.air.com.EasyRandomVideoChat
skyhookwireless.com	app.local1285
skyhookwireless.com	appinventor.ai_malote1971.SpainParanormalKII
skyhookwireless.com	app.qrcode
skyhookwireless.com	com.abtnprojects.ambatana
skyhookwireless.com	air.com.touchmultimedia.comicpuppetsfree
skyhookwireless.com	a2z.Mobile.Event4164
smaper.com	com.smaper.artisto

Table 7. Second-level domains receiving large requests of at least 100 KB and the apps that generated them.

7.1 Appsee: Screen Recording

Our first case study focuses on a video leak from the GoPuff app (`com.main.gopuff`) referenced in Table 5. The app provides on-demand delivery for users. The video was leaked to a third-party domain `api.appsee.com` that is owned by Appsee [2], an app analytics platform provider. They offer the ability to "[w]atch every user action and understand exactly how they use your app, which problems they're experiencing, and how to fix them. See the app through your users' eyes to pinpoint usability, UX and performance issues." [2] As we discuss below, this claim is—much to the chagrin of user privacy—accurate.

We began by decompiling the APK for GoPuff, which revealed that GoPuff starts Appsee as soon as the app launches (using the code in Figure 3). Our dynamic analysis confirmed this: as soon as a user opens GoPuff, the app records the screen and sends a video of this interaction to the following domain: `https://c6e83853bc68d0b076811737cb58920b.api.appsee.com/upload`. Taking a recording of user in-

```
package
com.main.gopuff.presentation.view.activities;

public class SplashActivity extends BaseActivity
implements SplashScreenView {
  // The method onCreate is called when
  // SplashActivity is created
  public void onCreate(Bundle paramBundle) {
    Appsee.start(getString(2131296433));
    ...
  }
}
```

Fig. 3. Code snippet from GoPuff, which uses the Appsee library to recode the screen as a user interacts with the app. The recording starts immediately when the user opens the app, and in some cases include users' PII (which is shared with Appsee).

Appsee Method	# of Apps	# of Occurrences
start	16	37
addEvent	7	27
setUserId	6	6
markViewAsSensitive	5	44
startScreen	4	9
stop	2	2
resume	1	6
pause	1	1
set3rdPartyId	1	1
Total	21	133

Table 8. Number of apps using various methods of the Appsee library, and how often they called each method.

teractions is not itself necessarily a privacy risk. However, even in this simple example we found that PII was exposed to Appsee—in this case the user's ZIP code.[2]

While this specific example exposes relatively low-risk PII, it is important to reiterate that Appsee requires no special permission to record the screen, nor does it notify the user that she is being recorded. In fact, Appsee puts the burden on the app developer to protect sensitive information by calling `markViewAsSensitive` in the app's code, or using server-side configuration through Appsee's dashboard [3].

At first glance, this is good news: the developer is in the position of knowing what views in their app are sensitive. However, our analysis indicates that many developers either have no sensitive data input, or simply did not bother to mark any view as sensitive: only five out of 33 apps in our dataset that include Appsee even call the `markViewAsSensitive` method. We show counts of other method calls in Table 8; most apps start recording (16 `start` and four `startScreen`), but only a small fraction of apps made calls to the `stop`/`pause` actions. Thus, in many cases screen recording is started, never stops, and no views are omitted from recording using the client-side AppSee API. It is unknown how many app developers use AppSee's dashboard to filter sensitive views on the server-side.

Screen recording, if adopted at scale and/or in apps that handle sensitive data, could expose substantial amounts of users' PII, especially when the full burden of securing private information is placed on developers. Further, we argue that the recording of interactions with an app (without user knowledge) is itself a privacy violation akin to recording audio or video of the user.

2 We disclosed this to GoPuff, which in response pulled the Appsee SDK from their iOS and Android apps and updated their privacy policy [12].

Given the risks of screen recording, we disclosed this behavior to Google's privacy team. Their response was that "Google constantly monitors apps and analytics providers to ensure they are policy-compliant. When notified of our findings, they reviewed GoPuff and AppSee and took the appropriate actions."

7.2 TestFairy: Screenshots

Our next case study focuses on a similar privacy risk: taking screenshots of the app while in use. TestFairy [25] is a mobile beta-testing platform that records user interactions via screenshots. In our dataset, SAHIC (`com.allintheloop.sahic`), which is a networking app for two conferences – SAHIC Cuba and SAHIC South America 2017 – uses the library and sent 45 screenshots to `testfairy.com`. The screenshots, shown in Figure 4, include (but are not limited to) information such as a search for attendees, a message to a contact, and a response to a survey. Attached with the screenshots is information that describes the current view and activity name of the app as shown in the following request:

```
https://collector-10.testfairy.com/services/
  ?method=testfairy.session.addScreenshot\
  &timestamp=1504971161996\&seq=1\
  &sessionToken=80775553-4252621-5418832-376287176
  -bab9f09e42c3c2e13a083c070ca30ed203aa05b6\
  &lastScreenshotTime=349\&interval=2000\&type=0\
  &activityName=com.allintheloop.sahic.MainActivity
```

While this feature is typically used during beta testing, the app was *not* labeled as a beta version in the Google Play Store. The user is also not informed of the recording, nor is she offered the opportunity to consent to beta testing upon opening the app. Thus, any reasonable user of these apps would likely never expect screenshots of her interactions.

To understand how pervasive this problem is, we examine all the apps in our dataset that include the TestFairy library. Fortunately, we found only one (SAHIC) out of 16 apps calling any of the TestFairy API methods for screenshots, and this is consistent with our network traffic analysis. Thus, despite a substantial privacy risk from this feature, we find that nearly all apps we tested are properly removing TestFairy methods before releasing their apps in the Google Play Store.

7.3 Photo Apps: Unexpected Sharing

Many users regularly use the cameras on their phones to take photos for personal use, then edit those photos using apps installed on their phones. In fact, both Android and iOS already provide powerful built-in ways to edit photos directly on the phone. That said, there is also a marketplace of photography apps that provide photo-editing features (e.g., filters, adding text, etc.). It is reasonable for most users to assume that such editing is performed on the device itself; however, we observed that several photography apps send the photos to their servers for processing without explicitly notifying users.

An example of this behavior is *Photo Cartoon Camera - PaintLab* (`com.fotoable.paintlab`), which uploads to their servers any photo that a user selects for editing, as well as any photo taken from the app (even before the user decides to edit the photo). Given that nothing else in the app indicates the need for an internet connection, the behavior is unexpected. Further, uploading photos taken from within the app before users decide to keep them exposes those users to further privacy risks from unintentional photo sharing. This behavior also appears in *InstaBeauty - Makeup Selfie Cam* (`com.fotoable.fotobeauty`), an app from the same developer, and in five other photo-editing apps.

We crawled the categories of 8,689 unique apps in our dataset that were from the Google Play Store. Our crawler was able to identify the categories of 7,022 apps. Out of those 7,022 apps, 463 apps were part of the "Photography" category. Our experiments detected 6 apps exhibiting this uploading behavior.

The privacy disclosures for these apps are not entirely clear. *Fotoable*, the developer of two aforementioned apps, has a privacy policy disclosure that makes only a general statement that personal information may be collected and used [10]. Three other apps, *FaceApp* (`io.faceapp`), *Picas - Art Photo Filter, Picture Filter* (`com.picas.photo.artfilter.android`), and *Prisma Photo Editor* (`com.neuralprisma`) specifically include users' photos as "personal information" collected [7, 19, 21]. However, this disclosure is arguably misleading as the app does not indicate uploading of a user's photo while they are editing it. In one app, *Artisto - Video & Photo Editor* (`com.smaper.artisto`), the privacy policy does not even seem to apply to this app—rather, it appears to be a general privacy policy for the developer's family of apps, and is focused on games [17]. Thus, it is reasonable to assume that users of these apps may not be aware of photo exfiltration and may not have consented to it.

8 Limitations

We now discuss some important issues and limitations of our study. From a set of 17,260 apps, we uncovered few instances of covert recording (i.e. apps taking pictures or videos without users intentionally doing so). On the one hand, this is good news: a very large fraction of apps are not abusing the ability to record media. On the other hand, it could also indicate that our analysis missed other cases of media leaks.

Dynamic analysis limitations. A number of factors could lead to this result. *First*, our media extraction method is not perfect. For example, an app could transform an audio recording into a different format (e.g., a text transcript or musical features such as beat and notes) that our system does not detect. Similarly, our approach does not stitch together a single media file transferred over multiple flows, or cases where a media file does not use a standard encoding format. *Second*, we may miss cases where multiple apps collude to subvert the permission model, e.g., when an app uses an Intent to launch another app [35]. *Third*, we do not detect media that is intentionally obfuscated when it is sent over the network, or encrypted at the application-layer (Mitmproxy does enable us to bypass TLS encryption).

It is possible for automated interactions to trigger a legitimate media exfiltration that could be mistakenly classified as a media *leak*. To mitigate this issue, we regularly captured screenshots during the automated interactions, then manually verified that a media leak was not generated by an intentional trigger in the app, e.g., camera shutter or audio recording button.

Static analysis limitations. We used static analysis to identify apps that *might* record media, namely by identifying corresponding API calls. It is well known, however, that the existence of an API call in a piece of

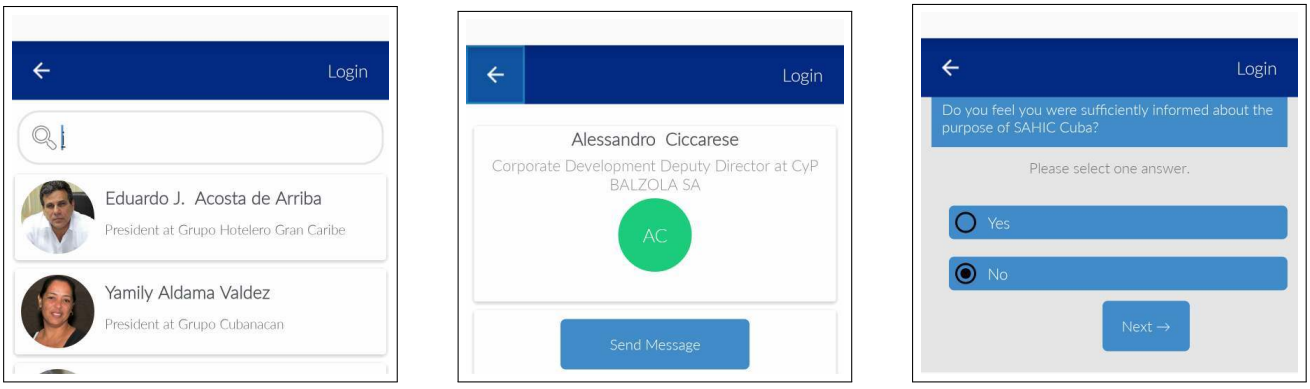

Fig. 4. Example screenshots collected by TestFairy. *Left*: Contact info. *Center*: Messaging another user. *Right*: Responses to a survey.

code does not guarantee it will ever be executed. To address this, we used dynamic analysis to filter out false positives. However, this does not address false negatives (where media API calls are reachable, but our automated interaction tool does not trigger them).

Further, our static analysis approach focuses on methods from the Android SDK and not native code, so we may miss cases of media leaks. Likewise, we may miss leaks from dynamically loaded code.

We rely on LibRadar and LibScout to identify third-party libraries. However, these tools may not be able to detect obfuscated libraries, or new versions of previously identified libraries. Fortunately, these limitations did not hinder our ability to identify the sources of media leaks in our study.

Future work. There are several ways to address the above issues. More sophisticated static analysis approaches could determine whether referenced methods are reachable during normal interactions with an app. A better understanding of how media may be sent over the network, and potentially transformed before transmission, would reduce our false negative rate. Our analysis could also incorporate analysis of native code that leaks media recordings.

Lastly, while we focused our analysis on Android apps, we will investigate in future work whether iOS apps exhibit similar behavior, as e.g., AppSee and TestFairy also provide iOS SDKs.

9 Conclusion

In this paper, we investigated the potential for, and specific instances of, multimedia recordings being sent over the internet by 17,260 popular Android apps across multiple app stores. We find that several apps leak content recorded from the camera and the screen over the internet, and in ways that are either undisclosed or unexpected given the purpose of the app. Importantly, we find that third-party libraries record a video of a user's interaction with an app, including at times sensitive input fields, *without any permissions or notification to the user*. Further, several apps share users' photos and other media over the internet without explicitly indicating this to the user. We also find that there is poor correlation between the permissions that an app *requests* and the permissions that an app *needs* to successfully run its code. This opens up the potential for unexpected exposure to additional media exfiltration with the inclusion of new libraries in future versions of the app. In ongoing work, we are continuing to monitor how multimedia content leaks over the internet from mobile and IoT devices, and assess the implications of such behavior.

Acknowledgments

We thank the anonymous reviewers and our shepherd Joel Reardon for their valuable feedback.

This material is based upon work supported by the DHS S&T contract FA8750-17-2-0145; the NSF under Award No. CNS-1408632, IIS-1408345, and IIS-1553088; a Security, Privacy and Anti-Abuse award from Google; a Comcast Innovation Fund grant; and a Data Transparency Lab grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Apache Thrift. https://thrift.apache.org/.

[2] Appsee Mobile App Analytics. https://www.appsee.com/.

[3] Appsee Tutorials: Protecting Users' Privacy. https://www.appsee.com/tutorials/privacy. (last accessed 06/14/2018).

[4] Autopsy. https://www.sleuthkit.org/autopsy/.

[5] CalOPPA Chapter 22: Internet Privacy Requirements. https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?lawCode=BPC&division=8.&title=&part=&chapter=22.&article=.

[6] dex-method-list. https://github.com/JakeWharton/dex-method-list.

[7] FaceApp Privacy Policy. http://archive.today/2018.06.14-232005/https://www.faceapp.com/privacy. (last accessed 06/14/2018).

[8] Fair Information Practice Principles (FIPPS). https://www.dhs.gov/sites/default/files/publications/consolidated-powerpoint-final.pdf.

[9] Foremost. http://foremost.sourceforge.net/.

[10] Fotoable Privacy Policy. http://archive.today/2018.06.14-230916/https://www.fotoable.com/privacy.html. (last accessed 06/14/2018).

[11] General Data Protection Regulation (GDPR). https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679&from=EN.

[12] GoPuff Privacy Agreement. https://gopuff.com/privacy-agreement. (last accessed 06/14/2018).

[13] JustTrustMe. https://github.com/Fuzion24/JustTrustMe.

[14] LaZy_NT. https://pypi.python.org/pypi/LaZy_NT.

[15] Mediaextract. https://github.com/panzi/mediaextract.

[16] Mitmproxy. https://mitmproxy.org/.

[17] My.com Terms of Use. http://archive.today/2018.06.14-231903/https://legal.my.com/us/games/tou/. (last accessed 06/14/2018).

[18] PhotoRec. https://www.cgsecurity.org/wiki/PhotoRec.

[19] Picas.tech Privacy Policy. http://archive.today/2018.06.14-231220/https://www.picas.tech/privacyandroid.php. (last accessed 06/14/2018).

[20] PIL. https://pypi.python.org/pypi/PIL.

[21] Prisma Privacy Policy. http://archive.today/2018.06.14-232142/http://prisma-ai.com/privacy.html. (last accessed 06/14/2018).

[22] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[23] Scalpel. https://github.com/sleuthkit/scalpel.

[24] tcpxtract. http://tcpxtract.sourceforge.net/.

[25] TestFairy Mobile Testing Platform. https://www.testfairy.com/.

[26] UI/Application Exerciser Monkey. https://developer.android.com/tools/help/monkey.html.

[27] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proc. of the International Conference on Mining Software Repositories (MSR), 2016.

[28] Daniel Arp, Erwin Quiring, Christian Wressnegger, and Konrad Rieck. Privacy Threats through Ultrasonic Side Channels on Mobile Devices. In Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P), 2017.

[29] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2014.

[30] Michael Backes, Sven Bugiel, and Erik Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In Proc. of the ACM Conference on Computer and Communications Security (CCS), 2016.

[31] Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. "Little Brothers Watching You:" Raising Awareness of Data Leaks on Smartphones. In Proc. of the Symposium on Usable Privacy and Security (SOUPS), 2013.

[32] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. In Proc. of the IEEE Mobile Security Technologies Workshop (MoST), 2013.

[33] Theodore Book and Dan S. Wallach. A Case of Collusion: A Study of the Interface Between Ad Libraries and Their Apps. In Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2013.

[34] Justin Brookman, Phoebe Rouge, Aaron Alva, and Christina Yeung. Cross-Device Tracking: Measurement and Disclosures. In Proc. of the Privacy Enhancing Technologies Symposium (PETS), 2017.

[35] Paolo Calciati and Alessandra Gorla. How do Apps Evolve in Their Permission Requests? A Preliminary Study. In Proc. of the International Conference on Mining Software Repositories (MSR), 2017.

[36] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2015.

[37] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In Proc. of the International Conference on Financial Cryptography and Data Security (FC), 2016.

[38] Terence Chen, Imdad Ullah, Mohamed Ali Kaafar, and Roksana Boreli. Information Leakage through Mobile Analytics Services. In Proc. of the ACM Workshop on Mobile Computing Systems and Applications (HotMobile), 2014.

[39] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated Test Input Generation for Android: Are We There Yet? In Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015.

[40] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2017.

[41] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. NetworkProfiler: Towards Automatic Fingerprinting of Android Apps. In Proc. of IEEE International Conference on Computer Communications (INFOCOM), 2013.

[42] Anupam Das, Nikita Borisov, and Matthew Caesar. Do You Hear What I Hear?: Fingerprinting Smart Devices Through Embedded Acoustic Components. In Proc. of the ACM Conference on Computer and Communications Security (CCS), 2014.

[43] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.

[44] Steven Englehardt. No boundaries: Exfiltration of personal data by session-replay scripts. https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/, November 2017.

[45] Tobias Fiebig, Jan Krissler, and Ronny Hänsch. Security Impact of High Resolution Smartphone Cameras. In Proc. of the USENIX Workshop on Offensive Technologies (WOOT), 2014.

[46] Jessica Fridrich. Sensor Defects in Digital Image Forensic. In Digital Image Forensics, pages 179–218. Springer, 2013.

[47] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In Proc. of the International Conference on Trust and Trustworthy Computing (TRUST), 2012.

[48] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In Proc. of the International Conference on Software Engineering (ICSE), 2013.

[49] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys), 2014.

[50] Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, University of Maryland, College Park, 2012.

[51] Michael Kassner. Take secret photos by exploiting Android's camera app. https://www.techrepublic.com/article/take-secret-photos-by-exploiting-androids-camera-app/, June 2014.

[52] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. In Proc. of the IEEE Mobile Security Technologies Workshop (MoST), 2012.

[53] Tadayoshi Kohno, Andre Broido, and KC Claffy. Remote Physical Device Fingerprinting. IEEE Transactions on Dependable and Secure Computing, 2(2):93–108, 2005.

[54] Anh Le, Janus Varmarken, Simon Langhoff, Anastasia Shuba, Minas Gjoka, and Athina Markopoulou. AntMonitor: A System for Monitoring from Mobile Devices. In Proc. of the ACM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data (C2B(1)D), 2015.

[55] Christophe Leung, Jingjing Ren, David Choffnes, and Christo Wilson. Should You Use the App for That?: Comparing the Privacy Implications of App- and Web-based Online Services. In Proc. of the Internet Measurement Conference (IMC), 2016.

[56] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In Proc. of the International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014.

[57] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In Proc. of the International Conference on Software Engineering (ICSE), 2016.

[58] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In Proc. of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2013.

[59] Vasilios Mavroudis, Shuang Hao, Yanick Fratantonio, Federico Maggi, Giovanni Vigna, and Christopher Kruegel. On the Privacy and Security of the Ultrasound Ecosystem. In Proc. of the Privacy Enhancing Technologies Symposium (PETS), 2017.

[60] Giuseppe Petracca, Yuqiong Sun, Trent Jaeger, and Ahmad Atamli. AuDroid: Preventing Attacks on Audio Channels in Mobile Devices. In Proc. of the Annual Computer Security Applications Conference (ACSAC), 2015.

[61] Ashwin Rao, Arash Molavi Kakhki, Abbas Razaghpanah, Anke Li, David Choffnes nad Arnaud Legout, Alan Mislove, and Phillipa Gill. Meddle: Enabling Transparency and Control for Mobile Internet Traffic. Journal of Technology Science (JoTS), (2015103003), October 2015.

[62] Ashwin Rao, Arash Molavi Kakhki, Abbas Razaghpanah, Amy Tang, Shen Wang, Justine Sherry, Phillipa Gill, Arvind Krishnamurthy, Arnaud Legout, Alan Mislove, and David Choffnes. Using the Middle to Meddle with Mobile. Technical Report NEU-CCS-2013-12-10, Northeastern University, 2013.

[63] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Christian Kreibich, and Phillipa Gill. Apps, Trackers, Privacy, and Regulators: A Global Study of the Mobile Tracking Ecosystem. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2018.

[64] Jingjing Ren, Martina Lindorfer, Daniel Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. Bug Fixes, Improvements, ... and Privacy Leaks – A Longitudinal Study of PII Leaks Across Android App Versions. In Proc. of the Network and Distributed System Security Symposium (NDSS), 2018.

[65] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. ReCon: Revealing and Controlling Privacy Leaks in Mobile Network Traffic. In Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys), 2016.

[66] Irwin Reyes, Primal Wiesekera, Joel Reardon, Amit Elazari Bar On, Abbas Razaghpanah, Narseo Vallina-

Rodriguez, and Serge Egelman. "Won't Somebody Think of the Children?" Examining COPPA Compliance at Scale. In Proc. of the Privacy Enhancing Technologies Symposium (PETS), 2018.

[67] Animesh Shrivastava, Puneet Jain, Soteris Demetriou, Landon P. Cox, and Kyu-Han Kim. CamForensics: Understanding Visual Privacy Leaks in the Wild. In Proc. of the ACM Conference on Embedded Networked Sensor Systems (SenSys), 2017.

[68] Szymon Sidor. Exploring limits of covert data collection on Android: apps can take photos with your phone without you knowing. http://www.ez.ai/2014/05/exploring-limits-of-covert-data.html, May 2014.

[69] Yihang Song and Urs Hengartner. PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices. In Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), 2015.

[70] Aatif Sulleyman. Facebook could secretly watch users through webcams, patents reveal. http://www.independent.co.uk/life-style/gadgets-and-tech/news/facebook-plans-to-watch-users-through-webcams-spy-patent-application-social-media-a7779711.html, June 2017.

[71] Vincent F. Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P), 2016.

[72] Narseo Vallina-Rodriguez, Jay Shah, Alessandro Finamore, Hamed Haddadi, Yan Grunenberger, Konstantina Papagiannaki, and Jon Crowcroft. Breaking for Commercials: Characterizing Mobile Advertising. In Proc. of the Internet Measurement Conference (IMC), 2012.

[73] Narseo Vallina-Rodriguez, Srikanth Sundaresan, Abbas Razaghpanah, Rishab Nithyanand, Mark Allman, Christian Kreibich, and Phillipa Gill. Tracking the Trackers: Towards Understanding the Mobile Advertising and Tracking Ecosystem. In Proc. of the Workshop on Data and Algorithmic Transparency (DAT), 2016.

[74] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. Orlis: Obfuscation-Resilient Library Detection for Android. In Proc. of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2018.

[75] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective Real-time Android Application Auditing. In Proc. of the IEEE Symposium on Security and Privacy (S&P), 2015.

[76] Ning Xia, Han Hee Song, Yong Liao, Marios Iliofotou, Antonio Nucci, Zhi-Li Zhang, and Aleksandar Kuzmanovic. Mosaic: Quantifying Privacy Leakage in Mobile Networks. In Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), 2013.

[77] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In Proc. of the USENIX Security Symposium, 2012.

[78] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In Proc. of the ACM Conference on Computer and Communications Security (CCS), 2013.

[79] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In Proc. of the ACM Conference on Computer and Communications Security (CCS), 2013.

[80] Zhe Zhou, Wenrui Diao, Xiangyu Liu, and Kehuan Zhang. Acoustic Fingerprinting Revisited: Generate Stable Device ID Stealthily with Inaudible Sound. In Proc. of the ACM Conference on Computer and Communications Security (CCS), 2014.

[81] Sebastian Zimmeck, Jie S. Li, Hyungtae Kim, Steven M. Bellovin, and Tony Jebara. A Privacy Analysis of Cross-device Tracking. In Proc. of the USENIX Security Symposium, 2017.