HideMyApp : Hiding the Presence of Sensitive Apps on Android

Anh Pham
EPFL, Switzerland

Italo Dacosta
EPFL, Switzerland

Eleonora Losiouk
University of Padova, Italy

John Stephan
EPFL, Switzerland

Kévin Huguenin
University of Lausanne, Switzerland

Jean-Pierre Hubaux
EPFL, Switzerland

Abstract

Millions of users rely on mobile health (mHealth) apps to manage their wellness and medical conditions. Although the popularity of such apps continues to grow, several privacy and security challenges can hinder their potential. In particular, the simple fact that an mHealth app is installed on a user's phone can reveal sensitive information about the user's health. Due to Android's open design, any app, even without permissions, can easily check for the presence of a specific app or collect the entire list of installed apps on the phone. Our analysis shows that Android apps expose a significant amount of metadata, which facilitates fingerprinting them. Many third parties are interested in such information: Our survey of 2917 popular apps in the Google Play Store shows that around 57% of these apps explicitly query for the list of installed apps. Therefore, we designed and implemented `HideMyApp` (`HMA`), an effective and practical solution for hiding the presence of sensitive apps from other apps. `HMA` does not require any changes to the Android operating system or to apps yet still supports their key functionalities. By using a diverse dataset of both free and paid mHealth apps, our experimental evaluation shows that `HMA` supports the main functionalities in most apps and introduces acceptable overheads at runtime (*i.e.,* several milliseconds); these findings were validated by our user-study ($N = 30$). In short, we show that the practice of collecting information about installed apps is widespread and that our solution, `HMA`, provides a robust protection against such a threat.

1 Introduction

Mobile health (mHealth), the use of technologies such as smartphones and wearable sensors for wellness and medical purposes, promises to improve the quality of and reduce the costs of medical care and research. An increasing number of people rely on mHealth apps to manage their wellness and to prevent and manage diseases.[1] For instance, more than a third of physicians in the US recommend mHealth apps to their patients [23], and there are around 325,000 mHealth apps available in major mobile app stores.[2]

Given the sensitivity of medical data, the threats of privacy leakage are one of the main hindrances to the success of mHealth technologies [37]. In this area, a serious and often overlooked threat is that an adversary can infer sensitive information simply from the presence of an app on a user's phone. Previous studies have shown that private information, such as age, gender, race, and religion, can be inferred from the list of installed apps [22, 29, 46]. With the increasing popularity of mHealth apps, an adversary can now infer even more sensitive information. For example, learning that a user has a diabetes app reveals that the user probably suffers from this disease; such information could be misused to profile, discriminate, or blackmail the user. When inquired about this threat, 87% of the participants in our user-study expressed concern about it (Section 10.6).

Due to Android's open design, a zero-permission app can easily infer the presence of specific apps, or even collect the full list of installed apps on the phone [54]. Our analysis shows that Android exposes a considerable amount of static and runtime metadata about installed apps (Section 4); this information can be misused by a nosy app to accurately fingerprint these apps. In 2014, Twitter was criticized for collecting the list of installed apps in order to offer targeted ads.[3] But Twitter is not the only app interested in such information. Our static and dynamic analysis of 2917 popular apps in the US Google Play Store shows that approximately 57% of these apps include calls to API methods that explicitly collect the list of installed apps (Section 5). Our analysis, corroborating the findings of previous studies [29, 32], also shows that free apps are more likely to query for such information and that third-party libraries (libs) are the main requesters of the list of installed apps. As users have on average 80 apps installed on their phones,[4] most of them being free, there is a high chance of untrusted third-parties obtaining the list of installed apps.

Since 2015, Android has classified as potentially harmful apps (PHA)[5] the apps that collect information about other apps without user consent [1]. To avoid this classification, developers simply need to provide a privacy policy that describes how the app collects, uses, and shares user data.[6] We find it interesting that only 7.7% of the evaluated apps clearly declared that they collect the list of installed apps in their privacy policies, and some even claim that such a list is non-personal information (Section 5.4). Also, few users read privacy policies [41], as our user study also confirmed (Section 10.6).

Android does not provide mechanisms to hide the use of sensitive apps on a phone; a few third-party tools, designed for other purposes, can provide only partial protection to some users (Section 6). Android announced that their security services will display warnings on apps that collect without consent users' personal information, including the list of installed apps.[7] This is a welcomed step, but the effectiveness of security warnings is known to be limited [30, 48] and it is unclear how queries by third-party libraries will be handled. It is also unclear if such an approach will be able to prevent more subtle attacks, where a nosy app checks for the existence of a specific app or a small set of sensitive apps by using more advanced fingerprinting techniques (Section 4).

We propose `HideMyApp` (`HMA`), the first system that enables organizations and developers to distribute sensitive apps to their users while considerably reducing the risk of such apps being detected by nosy apps on the same phone. Apps protected by `HMA` expose significantly less identifying metadata, therefore, it is more difficult for nosy apps to detect their presence, even when the nosy apps have all Android permissions and debugging privileges. With `HMA`, an organization such as a consortium of hospitals sets up an `HMA` app store where authorized developers collaborating with the hospitals can publish their mHealth and other sensitive apps. Users employ a client app called `HMA Manager` to anonymously (un)install, use, and to update the apps selected from the `HMA` app store; an the `HMA App Store` does not learn about the set of apps that a user has installed from the store. `HMA` transparently works on stock Android devices, it does not require root access, and it preserves the app-isolation security model of the Android operating system (OS). Still, `HMA` preserves the key functionalities of mHealth apps, *e.g.,* connecting to external devices via Bluetooth, sending information over the Internet, and storing information in databases.

With `HMA`, users launch a sensitive app inside the context of a container app, without requiring the sensitive app to be installed. A container app is a dynamically generated wrapper around the Android application package (APK) of the sensitive app, and it is designed in such a way that the sensitive app cannot be fingerprinted yet still can support inter-process communication between the sensitive app and other installed apps. To launch the APK from the container app, `HMA` relies on techniques described in existing work: the dynamic loading of compiled source code and app resources from the APKs and user-level app-virtualization techniques, *e.g.,* [24, 25]. However, note that app virtualization alone is insufficient in providing robust protection against fingerprinting attacks, as many of the information leaks uncovered by our analysis are still possible when just app virtualization is used. Therefore, our main contribution is the design and evaluation of mechanisms built on top of app-virtualization in order to reduce the information leaks that could be exploited to fingerprint sensitive apps. `HMA` provides multiple tiers of protection: For baseline protection against current threats, `HMA` obfuscates static meta-data of sensitive apps (*e.g.,* their package names and components). To provide more advanced protection (*e.g.,* against side-channel attacks), HMA can add an additional layer of obfuscation for sensitive apps (*e.g.,* randomizing memory access), and in some cases, app developers might need to be involved to make changes to the apps. Moreover, we are the first to identify the security and functional limitations of using app virtualization for the purpose of hiding apps.

Our evaluation of `HMA` on a diverse set of both free and paid mHealth apps on the Google Play Store shows that `HMA` is practical, and that it introduces reasonable operational delays to the users. For example, in 90% of the cases, the delay introduced by `HMA` to the cold start of an mHealth app by a non-optimized proof-of-concept implementation of `HMA` is less than one second. At runtime, the delay introduced is of only several milliseconds. Moreover, our user-study ($N = 30$) suggests that `HMA` is user-friendly and of interest to users.

Our main contributions in this work are as follows.

- *Systemized knowledge*: We are the first to investigate the techniques that an app can use to fingerprint another app.[8] Also, through our static and dynamic analysis on apps from the Google Play Store, we gain understanding about the prevalence of the problem of apps fingerprinting other installed apps.

- *Design and implementation of a solution for hiding sensitive apps*: We present `HMA`, a practical system that provides robust defense against fingerprinting attacks that target sensitive apps on Android. `HMA` works on stock Android, and no firmware modification or root privilege is required.

- *Thorough evaluation of `HMA`*: The evaluation of `HMA`'s prototype on apps from the Google Play Store suggests that `HMA` is practical. Also, our user study suggests that `HMA` is perceived as usable. `HMA`'s source code is available at `https://github.com/lca1/HideMyApp`.

2 Related Work

Researchers have actively investigated security and privacy problems in the Android platform. Existing works show that third-party libs often abuse their permissions to collect users' sensitive information [35, 47], and that apps have suspicious activities *e.g.,* collecting call logs, phone numbers, and browser bookmarks [29, 42]. Zhou *et al.* [54] show that

Android's open design has made publicly available a number of seemingly innocuous phone resources, including the list of installed apps; these resources could be used to infer sensitive information about their users, *e.g.,* users' gender and religion [40, 45]. Similarly, Chen *et al.* [27] show how to fingerprint Android apps based on their power consumption profiles. A significant research effort has been devoted to fingerprinting Android apps based on their (encrypted) network traffic patterns [28, 50, 53]. Researchers have also shown that re-identification attacks are possible using a small subset of installed apps [22, 33]. Demetriou *et al.* [29], in the same line as our work, used static analysis to quantify the prevalence of the collections of the list of installed apps and their metadata by third-party libs. We go beyond their work, however, by systematically investigating all possible information leaks that nosy apps can exploit to fingerprint other apps and by performing a dynamic analysis and privacy-policy analysis.

Existing mechanisms for preventing apps from learning about the presence of another app are not sufficient (Section 6). As we will show in Section 8, user-level virtualization techniques that enable an app (called *target app*) to be encapsulated within the context of another app (called *container app*) can be used as a building block for `HMA`. These techniques are used to sandbox untrusted target apps (*e.g.,* [24,25]) or to compartmentalize third-party libs from the host apps (*e.g.,* [34]). As they were designed for a different problem, however, they do not directly help hide the presence of a sensitive target app: They either require the target app to be first installed, thus exposing them to nosy apps through public APIs, or they run multiple target apps inside the same container app, thus violating the Android's app-isolation security model. They also do not provide any insight into the possible information leaks that can be exploited to fingerprint apps and how their techniques can be used for hiding the presence of apps.

3 Background on Android

Android Security Model. Android requires each app to have a unique package name defined by its developers and cannot be changed during its installation or execution. Upon installation, the Android OS automatically assigns a unique user ID (UID) to each app and creates a private directory where only this UID has read and write permissions. Additionally, each app is executed in its dedicated processes. Thus, apps are isolated, or sandboxed, both at the process and file levels.

Apps interact with the underlying system via methods defined by the Java API framework and the shell commands defined by the Linux-layer interface. Some API methods require users to grant apps certain permissions. Android defines three main protection levels for apps: normal, signature, and dangerous permissions.[9] Apps can have special permissions; users are required to grant these permissions to apps through the Settings app. Any app can execute shell commands; however, depending on its privilege, *i.e.,* default app privilege, debugging (adb)[10] or root, the outputs of the same shell commands are different.

Android Apps and APK Files. An Android app must contain a set of mandatory information: a unique package name, an icon, a label, a folder containing resources, and at least one of the following components: activity, service, broadcast receiver and content provider. An activity represents a screen, and a service performs long-running operations in the background. A broadcast receiver enables an app to subscribe and respond to specific system-wide events. A content provider manages the sharing of data between components in the same app or with other apps. Apps can optionally support other features such as implicit or explicit intents, permissions, and some customized app configurations. Apps are distributed in the form of APK files. An APK is a signed zip archive that contains the compiled code and resources of the app. Each APK also includes a manifest configuration file, called `AndroidManifest.xml`; this file contains a description of the app (*e.g.,* its package name and components).

4 Fingerprintability of Android Apps

Here, we demonstrate that an app, depending on its capabilities (its granted permissions and/or privileges), can retrieve information about other installed apps. This includes static information (*i.e.,* information available after apps are installed and that typically does not change during apps' lifetimes), and runtime information (*i.e.,* information generated or updated by apps at runtime). Our analysis was conducted on Android 8.0. Its findings are summarized in Table 1.

Without Permissions. An app can easily check if a specific app is installed on the phone. This can be done by invoking two methods `getInstalledApplications()` and `getInstalledPackages()` (hereafter abbreviated as `getIA()` and `getIP()`, respectively); they return the entire list of installed apps. An app can also register broadcast receivers (*e.g.,* `PACKAGE_INSTALLED`) to be notified when a new app is installed. It can also use various methods of the `PackageManager` class (*e.g.,* `getResourcesForApplication()`) as an oracle to check for the presence of a specific app. These methods take a package name as a parameter and return *null* if the package name does not exist on the phone.

If Android restricts access to package names of installed apps (*e.g.,* by requiring permissions), an app can still retrieve other static information about installed apps for fingerprinting attacks. This includes their mandatory information: the names of their components, their icons, labels, resources, developers' signatures and signing certificates. This also includes custom features used by installed apps: their permissions, apps configurations (themes, styles, and supported SDK). Such information can be obtained through a number of methods in

	Without Permissions	**With Permissions**	**Default App Privilege**	**Debugging Privilege (adb)**
Static Information	*Core attributes:* + Package name + Component's names + Resources + Icon, label + Developers' signatures *Customizations:* ⋆ Permissions ⋆ Themes ⋆ Phone configurations	(*) See note	+ Package names	+ Package names + APK path + APK file
Runtime Information	None	*Dangerous Permissions:* ⋆ Files in external storage ◇ Network traffic *Special Permissions:* ◇ Storage consumption + Running processes - Layouts and their content	◇ UI states† ◇ Power consumption† ◇ Memory footprints†	⋆ Files in external storage ⋆ System log ⋆ System diagnostic outputs + Running processes ◇ Network consumption - Screenshots

Table 1: Identifying information about installed apps that an app can learn, w.r.t. its permissions and privileges, through the Java API framework and the Linux-layer kernel. Analysis was conducted on Android 8.0. Superscript † means that the information can be learnt only in older versions of Android (*e.g.,* Android 8.0 requires the calling app to have adb privilege). *(*) Note:* Granting permissions to a zero-permission app does not enable it to obtain more static information about other installed apps. The notations $+, \star$ and $\diamond$ indicate the resources that our system (`HMA`, see Section 8) can protect by default, by collaborating with app developers or by randomizing runtime information of the container apps, respectively. Resources marked with the $-$ sign cannot be protected by `HMA`.

the `PackageManager` class, *e.g.,* `getPackageInfo()`. Note that this can be done even when apps are installed with the forward-lock option enabled (option `-l` in the `adb install` command). We tested this in Android 6.0; Android 8.0 threw an exception for this `-l` option. A nosy app cannot retrieve the list of intent filters declared by other apps. However, it can learn the names of components of installed apps that can handle specific intent requests, by using methods such as `resolveActivity()`.

With Permissions. An app granted with the `READ_EXTERNAL_STORAGE` permission, a frequently requested dangerous-permission, can inspect for unique folders and files in a phone's external storage (a.k.a. SD card). Apps with VPN capabilities (permission `BIND_VPN_SERVICE`) can intercept network traffic of other apps; existing work shows that network traffic, even encrypted, can be used to fingerprint apps with good accuracy [49, 50, 53].

With special permissions, an app can obtain certain identifying information about other apps at their runtime. For instance, the `PACKAGE_USAGE_STATS` permission permits an app to obtain the list of running processes (method `getRunningAppProcesses()`), and statistics about network and storage consumption of all installed apps, including their package names, during a time interval (method `queryUsageStats()`). In addition, accessibility services[11] (with the `BIND_ACCESSIBILITY_SERVICE` permission) can have access to the layouts and the layouts' contents of other apps.

With Default App-Privilege. An app can retrieve the list of all package names on the phone. This can be done by obtaining the set of UIDs in the `/proc/uid_stat` folder and using the `getNameForUid()` API call to map a UID to a package name. An app can also infer the UI states (*e.g.,* knowing that another app is showing a login screen) [26], memory footprints (sequences of snapshots of the app's data resident size) [36] and power consumption [27] of other apps. Note that access to this information has been restricted in recent versions of Android (*e.g.,* Android 8.0 requires the app to have adb privilege).

With Debugging Privilege (adb). An app can retrieve the list of package names (command `pm list packages`) and learn the path to the APK file of a specific app (command `pm path [package name]`). Moreover, the adb privilege enables an app to retrieve the APK files of other apps (command `pull [APK path]`); the app can then use API methods such as `getPackageArchiveInfo()` to extract identifying information from the APK files. Also an app can learn about runtime behaviors of other apps by inspecting the system logs and diagnostic outputs (commands `logcat` and `dumpsys`). Moreover, with the adb privilege, apps can directly retrieve the list of running processes (command `ps`), take screenshots [38] or gain access to statistics about network usage of other apps (folder `/proc/uid_stat/[uid]`).

Our analysis shows that Android's open design exposes a significant amount of information that facilitates app-fingerprinting attacks. App developers themselves cannot obfuscate most of the aforementioned information for the purpose of hiding sensitive apps. For example, by design, the package name of an app is a global identifier in the Google Play Store. As a result, the obfuscation of apps' package names has to be done per user, *i.e.,* for each user, the same app needs to be uploaded to the Google Play Store with a different package name. Similarly, the names of the app's components also need to be obfuscated per user, hence this approach is not practical. To mitigate this risk, Android could follow an approach similar to iOS, *i.e.,* to remove or restrict API methods and OS resources that leak identifying information of apps. However, such an approach would be difficult to implement in Android, as most of these methods and resources have valid use cases and are widely used by apps. For instance, methods `getIA()` and `getIP()`, are used by many popular apps with millions of users, *e.g.,* launcher, security/antivirus, and storage/memory manager apps. Removing or restricting such methods would break many apps and anger both developers and users. Such an approach would also negatively affect the competitive advantages of Android, *i.e.,* its customizability and rich set of features, over iOS. In addition, restricting API methods would not solve the problem completely, as more subtle fingerprinting attacks would still be possible. For example, in iOS, the `canOpenURL()` method can be used to check if a particular app is on the phone. Since iOS 9.0, in order to have an arbitrarily high number of calls to this method, an app has to declare beforehand the set of apps that it wants to check. Otherwise, it can only call this method at most 50 times.[12] This restriction reduces the risks of fingerprinting attacks, but negatively affects both developers and users, *e.g.,* apps need to be updated frequently to update the list of apps. More importantly, even with 50 queries, a nosy app can still check if a specific app or small set of apps are installed on the phone.

A better approach is for Android to include a new "sensitive" flag that enables users to hide sensitive apps from other apps in the same phone, *i.e.,* other apps will not be able to use Android API methods to infer the existence of apps flagged as sensitive. Moreover, Android can include a new permission that users can grant to certain apps in order to enable these apps to detect apps flagged as sensitive. This approach, however, requires significant modifications and testing of Android's APIs, and it is unlikely to be adopted by Google in the short or medium term. Therefore, our goal is to design a solution that does not require changes to Android or sensitive apps and that can be available to users sooner.

5 Apps Inquiring about Other Apps

We analyze apps from the Google Play Store to estimate how common it is for apps to inquiry about other installed apps. Our analysis focuses on API calls that directly retrieve the list of installed apps (hereafter called `LIA`): `getIA()` and `getIP()`, because these two methods clearly show the intent of developers to learn about other apps, whereas the other methods presented in Section 4 can be used in valid use cases. Therefore, the results presented in this section is a *lower-bound* on the number of apps that fingerprint other apps.

5.1 Data Collection

We gathered the following datasets for our analysis.

APK Dataset. We collected APK files of popular free apps in the Google Play Store (US site). For each app category in the store (55 total), we gathered the 60 most popular apps. After eliminating duplicate entries, default Android apps, and brand-specific apps, we were left with 2917 apps.

Privacy-Policy Dataset. We collected privacy policies that corresponded to the apps in our dataset. Out of 2917 apps, we gathered 2499 privacy policies by following the links included in the apps' Google Play Store pages.

5.2 Static Analysis

For our static analysis, by using Apktool,[13] we decompiled the APKs to obtain their smali code, a human-readable representation of the app's bytecode. We searched in the smali code for occurrences of two methods `getIA()` and `getIP()`.[14] API calls can be located in three parts of the decompiled code: in the code of Android/Google libs and SDKs, in the code of third-party libs and SDKs, or in the code of the app itself. To differentiate among these three origins, we applied the following heuristic. First, methods found in paths containing the "com/google", "com/Android" or "Android/support" substrings, are considered part of Android/Google libs and SDKs. Second, methods found in paths containing the name of the app are considered part of the code of the app. We believe this is a reasonable heuristics, because package names of Android apps follow the Java package-name conventions with the reversed internet domain of the companies, generally two words long. If the methods do not match the first two categories, then they are considered part of the code of a third-party lib or SDK. Note that this approach, also used in previous work [29], cannot precisely classify obfuscated code or code in paths with no meaningful names. Such cases, however, represent only a small fraction in our analysis (less than 5%).

Table 2 shows the proportions of apps that invoke `getIA()` and `getIP()` w.r.t. different call origins. Of the 2917 apps evaluated, 1663 apps (57.0%) include at least one invocation of these two methods in the code from third-party libs and the apps. These results show a significant increase in comparison with the results presented in 2016 by Demetriou *et al.* [29]. These results also show that most sensitive requests come from third-party libs or SDKs; app developers might not be aware of this activity, as has been the case for other sensitive data such as location.[15]

Static analysis has two main limitations. First, methods appearing in the code might never be executed by the app. Second, it is possible that the sensitive methods do not appear in the code included in the APK, rather in the code loaded dynamically at runtime. To address these issues, we also performed a dynamic analysis of the apps in our dataset.

Analysis method	Call origin	`getIA()` (%)	`getIP()` (%)	`getIA()` or `getIP()` (%)
Static	Third-party libs + Apps	36.4	43.6	57.0
Static	Apps only	8.1	8.4	13.9
Dynamic	Third-party libs + Apps	6.5	15.0	19.2

Table 2: Proportion of free apps that invoke `getIA()` and `getIP()`, to collect `LIAs` w.r.t. different call origins.

5.3 Dynamic Analysis

For our dynamic analysis, by using XPrivacy[16] on a phone with Android 6.0, we intercepted the API calls from apps. For the analysis to scale, for each app, we installed it and granted it all the permissions requested. Next, we launched all the runnable activities declared by the app for 10 minutes. Although this approach has limitations, as it only has a short period of time per app and it cannot emulate all the activities a user could do, it is sufficient to estimate a *lower-bound* on the number of apps that query for `LIAs` at runtime, as shown in our results.

Our results, shown in Table 2, show that 190 apps (6.54%) called `getIA()`, 436 apps (15.0%) called `getIP()`, and 19.2% of the apps called at least one of these two methods. Because XPrivacy does not provide information about the origin of the request, we performed some additional steps. For each app, we used the results of our static analysis and searched for occurrences of `getIA()` and `getIP()` in the code belonging to Google/Android libs. We found that most apps did not include calls to these sensitive methods in the code belonging to Google/Android libs: 181 out of 190 for `getIA()` and 412 out of 436 for `getIP()`. Hence, we conclude that these sensitive requests came mainly from third-party libs or from the code of the apps.

Interestingly, we found 49 apps that called at least one of the two sensitive methods in our dynamic analysis, but not in our static analysis. This could be because the decompiler tool produced incorrect smali code, or because these requests were dynamically loaded at runtime. Still, this represents only a small number of the apps found through our analysis.

Our static and dynamic analysis shows that a significant number of free apps actively queries for `LIAs`: between 19.2% (dynamic analysis) and 57% (static analysis) of the tested apps.[17] This shows that many third parties are interested in knowing about the installed apps on users' phones, and that, if Android blocked `getIA()` and `getIP()`, they would likely attempt to use other methods (see Section 4).

5.4 Analysis of Privacy Policies

Google's privacy-policy guidelines require apps that handle personal or sensitive user data to comprehensively disclose how they collect, use and share the collected data. An example of a common violation, shown in these guidelines, is "*An app that doesn't treat a user's inventory of installed apps as personal or sensitive user data*".[18] Next, we explain what developers understand about the guidelines.

As mentioned in Section 5.1, out of 2917 apps in our dataset, we found 2499 privacy policies. From the 1674 nosy apps found in the static and dynamic analysis, 1524 apps have privacy policies. We semi-automated the policy analysis as follows. We built a set of keywords consisting of nouns and verbs that might be used to construct a sentence to express the intention of collecting `LIAs`: retrieve, collect, fetch, acquire, gather, package, ID, installed, app, name, application, software, and list. For each privacy policy, we extracted the sentences that contain at least one of the keywords. From the extracted sentences, we manually searched for specific expressions such as "installed app", "app ID" and "installed software". Thereafter, we read the matched sentences and the corresponding privacy policy.

From the set of 2499 policies, we found 162 policies that explicitly mention the collection of `LIAs`. Among these, 129 belong to the set of 1674 nosy apps (7.7%). Some apps have exactly the same privacy policies, even though they are from different companies (*e.g.,* [20] and [6]). 33 apps mentioned the collection of `LIAs`, but we did not find these apps in both static and dynamic analyses. For these apps, we performed a more thorough dynamic analysis: we used them as a normal user would, while intercepting API calls. We did not capture, however, any calls to the two sensitive methods. This might be because developers copy the privacy policies from other apps, or because the apps will make these calls in the future.

Besides the generic declared purposes of the collections of `LIAs` by apps, *e.g.,* for improving the service (*e.g.,* [14, 21]), some apps explicitly state that they collect `LIAs` for targeted ads (*e.g.,* [3, 12]), and targeted ads by third-party ad networks (*e.g.,* [15]). Unexpectedly, we found that of the 162 policies that mention the collections of `LIAs`, 76 categorize `LIAs` as *non-personal*, whereas Google defines this as personal information. This shows a misunderstanding between developers and Google's guidelines.

6 Existing Protection Mechanisms

To the best of our knowledge, there are no existing robust mechanisms for hiding sensitive apps. Below, we present some mechanisms that can offer partial protection.

6.1 Mechanisms by Google

Android does not provide users with a mechanism to hide the existence of apps from other apps. But users can repurpose existing Android mechanisms for partially hiding apps.

Multiple Users. Android supports multiple users on a single phone by separating user accounts and app data.[19] This feature could be used to prevent fingerprinting of sensitive apps by installing sensitive apps in one or more secondary accounts, thus isolating sensitive apps from nosy apps. However, a key disadvantage of using multiple users for this purpose is that it prevents inter-app communications (e.g., intent-based interactions) among apps in different user accounts. As a result, sensitive apps' functionalities can be significantly reduced because they cannot delegate tasks to other apps. For instance, a sensitive app will not have access to a user's calendar or contacts (unless the user replicates them on each account) or access to other apps for certain tasks, e.g., sending a message or picture via Whatsapp or Facebook, accessing files in Dropbox, sending an e-mail or SMS, and authenticating users with Google or Facebook accounts. In section 10.5, we show that popular mHealth apps use inter-app communications not only for delegating tasks but also for sharing their resources with other apps. Therefore, a solution that hides sensitive apps and that still supports inter-app communications is more desirable.

Multiple user accounts could also introduce new security and privacy issues [44]. Using multiple users will significantly affect the user experience, as users will have to switch back and forth among accounts to access different types of apps and data, introducing significant delays and confusion. While the primary account is in the foreground, apps on secondary accounts are put in the background and they cannot use Bluetooth services (important for mHealth apps). Another important problem is that some popular phone manufacturers (*e.g.,* Samsung, LG, Huawei, Asus) disable multiple users in some of their devices,[20] thus affecting the availability of this solution to many users.

We have also found experimentally that the implementation of multiple users in the latest (Android 9) and earlier versions of Android does not effectively prevent nosy apps from learning what other apps are installed in different user accounts. To bypass this protection, a nosy app could do any of the following:

- On Android 7 or earlier, including an additional parameter flag (`MATCH_UNINSTALLED_PACKAGES`) in methods `getIA()` and `getIP()` will reveal the apps installed in secondary user accounts.

- On Android 9 or earlier, a nosy app can use multiple `PackageManager` methods, such as `getPackageUid()`, `getPackageGidS()`, `checkPermission()`, `checkSignatures()`, or `getApplicationEnabledSetting()`, as oracles to check if an app is installed on a secondary account or on a work profile. The nosy app only needs to include the package name of the targeted sensitive app as a parameter to these methods. Android's source code shows that these methods check the user ID of the app calling the method to show only information of apps in the same user profile, but our experimental evaluation shows that currently deployed versions of Android do not enforce such checks. This approach was tested on Android 9.

- A nosy app can guess the UIDs of the apps installed on all the accounts and work profiles, by looking at the `/proc/uid` directory to learn the ranges of current UIDs in the system. It then guesses the UIDs of other apps and uses the `getNameForUid()` method to learn the package name. This method will return a package name given a UID as a input parameter; if the app does not exist, it returns null. As a result, it can be used as an oracle to retrieve the list of installed apps on the device. This was tested on Android 6, 8.1 and 9.

- A nosy app with adb privilege can easily verify if a sensitive app is running on the device, independently of the account or profile it was installed on, by using the shell command: *pidof <PackageName>*. This approach was tested on Android 9.

- A nosy app with adb privilege can obtain the list of installed apps, which includes apps on secondary accounts and work profiles, by using the shell command *dumpsys*. This approach was tested on Android 9.

Android for Work. Android supports an enterprise solution called Android for Work; this solution separates work apps from personal apps.[21] Our tests, using similar methods as with multiple users, also confirmed that, as with multiple users accounts, it is easy to identify which apps are in the work profile. In addition, Android for Work is only available to enterprise users.

Recently, Android introduced a new feature called *Instant Apps*;[22] this feature enables users to run apps instantly without installing them. Such an approach could be used to hide sensitive apps, however, it only supports a limited subset of permissions, and it does not support features that are crucial for mHealth apps such as storing users' data or connecting to Bluetooth-enabled devices.[23]

Google classifies the list of installed apps as *personal* information hence requires apps that collect this information to include in their privacy policies the purpose of their collection. Apps that do not follow this requirement are classified as Potentially Harmful Apps (PHAs) or Mobile Unwanted Softwares (MuWS) [1, 2]. Android security services, *e.g.,* Google Play Protect [10], periodically scan users' phones and warn users if apps behave as PHAs or MuWS. Such mechanisms, however, do not seem to effectively protect against the unauthorized collection of the list of installed apps. Our analyses show that only 7.7% of the apps declare their collections of such information in their privacy policies, and some claim that a list of installed app is non-personal information (Section 5).

Furthermore, these mechanisms might fail to detect targeted attacks, *e.g.,* a nosy app might want to check if a small subset of sensitive apps exists on the phone.

6.2 Mechanisms by Third Parties

Samsung Knox[24] relies on secure hardware to offer isolation between personal and work-related apps, similar to Android for Work. Unfortunately, we were able to evaluate the robustness of the protection offered by Knox w.r.t. hiding apps, because Samsung discontinued its support for work and personal spaces for private users; only enterprise users can use such a feature. Nevertheless, this solution is device specific and only hides apps from other apps in a different isolated environment, but not from apps in the same environment (apps in the same isolated environment can come from different, untrusted sources). That is, a solution that provides per-app isolation is preferable.

There are apps on the Google Play Store that help users to hide the icons of their sensitive apps from the Android app launcher (*e.g.,* [16]). Even though they help hide the presence of the sensitive apps from other human users (*e.g.,* nosy partners), these sensitive apps are still visible to other apps. Along the line of user-level virtualization techniques, on the Google Play Store, we found apps that use these techniques to enable users to run in parallel multiple instances of an app on their phones and to partially hide the app, (*e.g.,* [11, 17, 18]). However, these solutions require the hidden app to be installed first on the phone before protecting it, thus triggering installation and uninstallation broadcast events that can be detected by a nosy app. These apps provide only a single isolated space, *i.e.,* they do not protect apps from other apps in the same environment. Our preliminary evaluation of these apps also shows that their protection is limited, *e.g.,* the names of the hidden apps can be found in the list of running processes.

7 Our Solution: `HideMyApp`

We propose `HideMyApp` (`HMA`), a system for hiding the presence of sensitive apps w.r.t. to a nosy app on the same phone. In this section, we will present our system model, adversarial model, design goals and a high-level overview of the solution.

7.1 System Model

The scenario envisioned for `HMA` is as follows. A hospital or a hospital consortium (hereafter called hospitals) sets up an app store, called `HMA App Store`, where app developers working for the hospitals publish their mHealth apps. Hospitals want their patients to use their mHealth apps without disclosing their use to other apps on the same phone. Note that such organizations and their own app stores already exist, *e.g.,* the VA App Store set up by the U.S. Department of Veterans Affairs.

To enable the users to manage the apps provided by the `HMA App Store`, the `HMA App Store` provides the users with a client app called `HMA Manager`. This app can be distributed through any available app stores, *e.g.,* the Google Play Store. To allow the `HMA Manager` app to install apps downloaded from the `HMA App Store`, similarly to other Google Play Store alternatives *e.g.,* Amazon[25] and F-Droid [9], users need to enable the “allow apps from unknown sources" setting on their phones. Since Android 8.0, Google made this option more fine-grained by turning it into the “Install unknown apps" permission [19]. That is, users only need to grant this permission to the `HMA Manager` app to enable it to install apps from the `HMA App Store`.

7.2 Adversarial Model

We assume the Android OS on the user's phone to be trusted and secure, including its Linux kernel and its Java API framework. We assume that the `HMA App Store` and the `HMA Manager` app are trusted and secure, and that they follow the prescribed protocols of the system. We discuss mechanisms to relax the trust assumptions on the `HMA App Store` and `HMA Manager` app in Section 9.2.

We assume there is a nosy app that wants to learn if a specific app is present on the phone. The nosy app has the default app-privilege, and it is granted all dangerous permissions by its user – these are the typical capabilities of apps that users often install on their phones. In Section 9, we discuss mechanisms for preventing more advanced fingerprinting attacks by malicious apps; a malicious app has more capabilities than a nosy app, *i.e.,* it can have special permissions (*e.g.,* `PACKAGE_USAGE_STATS` or `BIND_ACCESSIBILITY_SERVICE`) and the debugging privilege (adb), thus it can perform more advanced attacks, such as fingerprinting apps using their runtime information.

We assume that apps belonging to hospitals are nosy, *i.e.,* these apps are also curious about what other apps are installed on the user's device.

7.3 Design Goals

The purpose of `HMA` is to effectively hide the presence of sensitive apps, yet preserve their usability and functionality.

- *(G1) Privacy protection.* It should be difficult for a nosy app to identify sensitive apps on the same phone.

- *(G2) No firmware modifications.* The solution should run on stock Android phones. That is, it should not require the phones to run customized versions of Android firmware, *e.g.,* extensions to Android's middleware or the Linux kernel. This also means that the solution should not require the phones to be rooted.

- *(G3) Preserving the app-isolation security model of Android.* Each app should have its own private directory and run in its own dedicated process.

- *(G4) Few app modifications.* For baseline protection against nosy apps, the solution should not require app developers to change their apps. For protection against malicious apps, apps might need to be changed or some features might not be supported.

- *(G5) Usability.* The solution should preserve the usability and the key functionalities of sensitive apps.

7.4 HMA Overview

From a high-level point of view, HMA achieves its aforementioned design goals by enabling its users to install a container app for each sensitive app (as illustrated in Fig. 1). Each container app has a generic package name and obfuscated app components. As a result, nosy apps cannot fingerprint a sensitive app by using the information about its container app. At runtime, the container app will launch the APK file of the sensitive app within its context by relying on user-level virtualization techniques. That is, the sensitive app is not registered in the OS.

To do so, HMA requires the hospitals to bootstrap the system by setting up the HMA App Store and distributing the HMA Manager app to users (Section 8.1 and 8.2). Through the HMA Manager app, users can (un)install, open, and update sensitive apps without being discovered by the OS and other apps. We detail these operations in Section 8.3.

8 HMA System Description

Here, we detail the components and operations of HMA.

8.1 HMA Manager App

Recall, to hide their presence, sensitive apps are not registered in the OS; instead, their container apps are registered. Thus, if users open their default Android app launcher, they will only see container apps with generic icons and random names. To solve this usability issue, at installation time, the HMA Manager app keeps track of the one-to-one mappings between sensitive apps and their container apps. Using the mappings, the HMA Manager app can display the container apps to the users with the original icons and labels of their sensitive apps. To provide unlinkability between users and their sensitive apps w.r.t. the HMA App Store, the HMA Manager app *never* sends any identifying information of the users to the HMA App Store, and all the communications between the HMA App Store and the HMA Manager are anonymous. This is a reasonable assumption because the HMA Manager app can be open-sourced and audited by third parties. Also, in most cases, users do not have fixed public IP addresses; they access the Internet via a NAT gateway offered by cellular providers. If needed, a VPN proxy or Tor could be used to hide network identifiers.

8.2 HMA App Store

The HMA App Store receives app-installation and app-update requests from HMA Manager apps and returns container apps to them. To reduce the delays introduced to the app-installation and app-update requests, the HMA App Store defines a set of P generic package names for container apps, *e.g.,* app-1, ..., app-P. This set of generic names is shared by all sensitive apps, thus there is no one-to-one mapping between a sensitive app and a generic name or a subset of generic names.[26] For each sensitive app, the HMA App Store can generate beforehand P container apps corresponding to P predefined generic package names and store them in its database. Below, we explain the procedure followed by the HMA App Store to create a container app. Details about the app-installation and update requests from the HMA Manager apps are explained in Section 8.3.

HMA Container-App Generation. To generate a container app for a sensitive APK, the HMA App Store performs the following steps. Note that this operation cannot be performed by the HMA Manager app, because Android does not provide tools for apps to decompile and compile other apps.

- The HMA App Store creates an empty app with a generic app icon, a random package name and label, and it imports into the app the lib and the code for the user-level virtualization, *i.e.,* to launch the APK from the container app. Note that the lib and the code are independent from the APK.

- The HMA App Store extracts the permissions declared by the sensitive app and declares them in the manifest file of the container app.

- To enable the container app to launch the sensitive APK, app components (activities, services, broadcast receivers, and content providers) declared by the sensitive app need to be declared in the manifest file of the container app. This information, however, can be retrieved by nosy apps to fingerprint sensitive apps (Section 4). To mitigate this problem, the container app declares activities, services and broadcast receivers of the sensitive app with random names. At runtime, the container app will map these random names to the real names. The intent filters declared in the components of sensitive apps are also declared in the manifest file of their sensitive apps. In Section 9, we will discuss the case of content providers.

- The HMA App Store compiles the container app to obtain its APK and signs it.

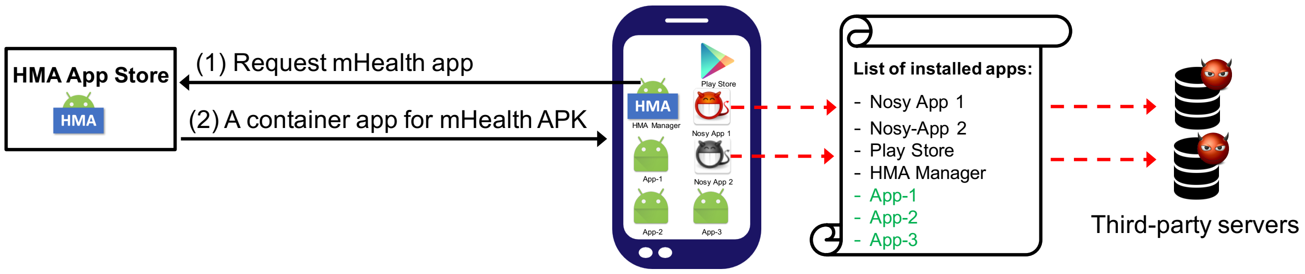

Figure 1: Overview of the `HMA` architecture. Nosy apps only learn the generic names of the container apps.

Note that for the sake of simplicity, here we only present a solution that protects mandatory features of Android apps. A malicious app might try to fingerprint sensitive apps based on, for instance, the runtime information produced by their container apps. We discuss this in Section 9.

`HMA` User-Level Virtualization. To launch the APK of a sensitive app without installing it, its container app generates a randomly named child-process in which the APK will run, *i.e.,* the APK is executed under the same UID as its container app. Thereafter, the container app loads the APK dynamically at runtime, and it intercepts and proxies the interactions between the sensitive app and the underlying system (the OS and the app framework). To do so, we rely on an open-source lib for app-virtualization called DroidPlugin [8].

8.3 `HMA` Operations

In this section, we detail the procedure followed by a user when she (un)installs, updates, or uses sensitive apps.

App Installation. To install a sensitive app, the user opens her `HMA Manager` app to retrieve the set of apps provided by the `HMA App Store`. Once she selects a sensitive app, the `HMA Manager` app sends to the `HMA App Store` an installation request consisting of the name of the sensitive app and her desired generic package name for the container app. The `HMA App Store` correspondingly finds in its database or creates a container app, and it sends the container app, together with the original label and icon of the sensitive app, to the `HMA Manager`. The `HMA Manager` prompts the user for her confirmation about the installation. Once the user accepts, the installation occurs as in standard app installation on Android. Also, the `HMA Manager` saves, in its private directory, a record of the package name of the container app and the package name, the original icon and the label of the sensitive app.

App Launch. To launch a sensitive app, the user opens her `HMA Manager` app to be shown with the set of container apps installed on her phone. Using the information stored in its database about the mappings between container apps and sensitive apps (Section 8.3), the `HMA Manager` displays to the user the container apps with the original labels and icons of the corresponding sensitive apps. Therefore, the user can easily identify and select her sensitive apps.

The *first time* a container app runs, it needs to obtain the sensitive APK from the `HMA App Store`; then it stores the APK in its private directory. This incurs some delays to the first launch of the sensitive app. However, it is needed to prevent the sensitive app from being fingerprinted: If the sensitive APK was included in the resources or assets folders of its container app so that the container app could copy and store the APK in its private directory at installation time, a nosy app would be able to obtain the sensitive APK. Recall, any app can obtain the resources and assets of other apps (Section 4). Also, Android does not permit apps to automatically start their background services upon installation.

At runtime, the container app dynamically loads the sensitive APK. Thereafter, it intercepts and proxies API calls and system calls between the sensitive app and the underlying system. If the version of the Android OS is at least 6.0, permissions requested by the sensitive app will be prompted by its container app at runtime. Thus, they will be shown with the generic package name of the container app. This, however, does not affect the comprehensibility of the permission requests, as shown by our user study (Section 10.6). Additionally, when an app sends an implicit intent with an `action` supported by the sensitive app, the operating system will show the sensitive app as an option for the user to choose to handle the requested action. This introduces a usability problem: the icon of the sensitive app presented to the user is a generic icon. This, however, can be solved by using the *direct share targets* feature in Android – a feature that enable apps to show finer-grained internal content in the *chooser dialog window*.[27]

App Update. When a sensitive app on the `HMA App Store` has an update, for each predefined generic container-app package name, the `HMA App Store` generates a corresponding container app for the updated sensitive app. This step is needed, because the configuration file of the container app needs to be updated w.r.t. the updates introduced by the sensitive app. The `HMA App Store` then sends a push notification to all `HMA Manager` clients to notify them about the update. If a user has the sensitive app on her phone, her `HMA Manager` sends the package name of its existing container app to the `HMA App Store`. In return, it receives the corresponding updated container app from the `HMA App Store`. It then prompts the user to confirm the installation. Once the user accepts, the updated container app is installed, similarly to the standard app-update procedure on Android.

App Uninstallation. To uninstall a sensitive app, the user opens her `HMA Manager` app to be shown with the set of her container apps. Once she selects the container app, the `HMA Manager` prompts her to confirm the uninstallation. Thereafter, the uninstallation occurs similarly to the standard app-uninstallation procedure on Android.

9 Privacy and Security Analysis

Here, we present an analysis of `HMA` to show that it effectively achieves its privacy and security goals w.r.t. different capabilities of the nosy apps (*i.e.,* their granted permissions and privileges) as shown in Table 1.

9.1 Privacy

Nosy Apps without Permissions. `HMA` effectively protects, by default, the core attributes of sensitive apps. First, a nosy app cannot obtain the package name of a sensitive app, because the sensitive app is never registered on the system; instead, its container app with a generic package name is installed. Second, the resources, shared libraries, developers' signatures and developers' signing certificates of the sensitive app cannot be learnt by the nosy app, because they are not declared or included in the container-app's APK; instead they are dynamically loaded from the sensitive APK at runtime. Third, the nosy app cannot learn the components' names of the sensitive app, because these names are randomized. To prevent fingerprinting attacks based on the number of components declared in the container app, the `HMA App Store` adds dummy random components during the generation of the container app such that all the container apps declare the same number of components.

A nosy app might try to fingerprint sensitive apps by using the sets of permissions declared by their container apps. This can be mitigated if all container apps declare a union of permissions requested by sensitive apps in the `HMA App Store`. Note that for devices with Android 6 or later, the container app requests at runtime only the permissions needed by its sensitive app, and users can grant or decline these requests. This makes it difficult for nosy apps to fingerprint a sensitive app using the set of permissions granted to its container app.

`HMA` needs collaboration from app developers to prevent fingerprinting attacks based on the customized configurations of some sensitive apps, *e.g.,* themes and screen settings. The `HMA App Store` can define a guideline for app developers to follow such that all apps have the same configurations. This will affect the look and feel of the sensitive apps, but it is a trade-off between usability and privacy. Note that the same approach has been used in other deployed systems, *e.g.,* in the Tor browser where all the versions have the same default window size and user-agent strings.[28] To facilitate guideline compliance, the `HMA App Store` can also provide developers with IDE plugins to help them write guideline-compliant code; such an approach has been proposed in existing work (*e.g.,* [43] and [31]).

App developers might want to use custom features, such as custom permissions, custom actions for the intent filters of their apps' components. These features, however, can be used to fingerprint their sensitive apps, hence should not be used by app developers. An app might want to support a content provider for sharing data between its components or for sharing data with other apps. `HMA` can support the former case; the container apps do not need to declare the content provider in its manifest file, but it handles the requests from the components of the sensitive apps internally. `HMA`, however, cannot support the case of sensitive apps using content providers to share data with other apps. This is because in order to do so, the container apps need to declare the URIs of their content providers in their manifest files, and these URIs can uniquely identify apps. These are limitations of `HMA`, however, from our analysis, only a small number of apps is affected by these limitations (Section 10.5).

Nosy Apps with Permissions. A nosy app can fingerprint sensitive apps based on their use of the external storage (SD card), *e.g.,* unique directories and files. To prevent this, container apps can intercept and translate calls from sensitive apps associated with the creation or access of files in external storage. However, note that apps are not recommended to store data there, especially mHealth apps. To prevent an app with VPN capabilities from fingerprinting sensitive apps based on the IP addresses in the header of the IP packages, the sensitive apps can relay their traffic through the `HMA App Store` servers; this protection is provided at the cost of additional communication delays for the apps and it requires collaboration with app developers.

A malicious app cannot fingerprint a sensitive app by using the list of running processes, because the sensitive app runs inside the child process of its container app with a random name. To prevent malicious apps from abusing its special permissions to fingerprint sensitive apps using their runtime statistics, *e.g.,* resources consumed by their container apps, the container apps can randomly generate dummy data to obfuscate the usage statistics of sensitive apps. Note that this does not require changes to the sensitive apps. In future work, we will evaluate techniques against these side-channel attacks such as [51] and [26]. `HMA` cannot prevent malicious apps, with permission to accessibility services, from fingerprinting sensitive apps. Accessibility services enable access to apps' unique layout information, and it is not practical to require all sensitive apps to use a generic layout. However, Google currently bans the use of accessibility services for purposes not related to helping users with disabilities[29]. Users should grant this permission only to apps they trust.

Nosy Apps with Default App Privileges. Recall, `HMA`, by default, hides the package name of the sensitive apps. To prevent nosy apps from fingerprinting sensitive apps by using their UI states, the container apps can also obfuscate the UI states by overlaying transparent frames on the real screens of the sensitive apps. Similarly to the case of other runtime statistics discussed above, the container apps can also randomly generate dummy data to obfuscate the memory footprints and power consumptions of the sensitive apps.

Malicious Apps with the Debugging Privilege (adb). Recall, HMA protects the package name and the process names of the sensitive apps by default. Also recall, the container apps can randomize runtime statistics of the container apps. In addition, the paths to the APK files of the container apps do not reveal any information about the sensitive apps. Also, the malicious app cannot retrieve the APK files of the sensitive apps, because the APKs are stored inside the private directories of their container apps.

To prevent advanced attacks by malicious apps, *e.g.,* fingerprinting sensitive apps by reading the log of the phone, HMA requires collaboration from app developers. Developers should not write identifying information about their sensitive apps to the log. Apps with adb privilege can take screenshots of the phone and infer apps' names from the screenshots. HMA cannot prevent this attack. However, note that this attack requires the malicious app to do extra and error-prone operations (*e.g.,* image processing) to identify sensitive apps.

9.2 Security

By using user-level virtualization techniques to launch an APK, HMA does not require users to modify the OS of the phone. The Android's app-isolation security model is also preserved, because each APK runs inside the context of its container app. Thus, it is executed in a process under the same UID as its container app, and it uses the private data directory of its container app. Similarly to other third-party stores (*e.g.,* Amazon or F-Droid), HMA requires users to enable the "allow apps from unknown sources" setting on their phones. However, apps installed from these sources are still scanned and checked by Android security services for malware [10]. Also, recently, this setting was converted to a per-app permission [19]. As a result, granting the HMA Manager app the permission to install apps from unknown sources will not give other apps on the phone the same permission.

As on the Google Play Store, with HMA, app developers register their public keys on the HMA App Store, and sign their apps before they submit to the HMA App Store. Moreover, the HMA App Store signs the container apps that it generates to vouch for the integrity of the container apps and the sensitive apps. This mechanism, however, introduces a security issue for sensitive apps: Apps from different developers are signed by the same private key of the HMA App Store, hence a dishonest app developer might exploit this same-signature property to access signature-protected components of other apps.[30] Note that requesting or declaring signature-protection permissions will facilitate fingerprinting of sensitive apps, hence HMA does not support this feature. As a result, this attack is not possible in HMA. Also note that few apps use signature-protected permissions (see Section 10.5). In future work, we will explore mechanisms for enabling container apps to verify the signatures of sensitive apps at runtime, in order to prevent unauthorized access to signature-protected components of their sensitive apps.

HMA container apps prompt users only for permissions requested by sensitive apps. To relax the trust assumptions on the HMA App Store and HMA Manager, the HMA App Store can provide an API so that anyone can implement her own HMA Manager app, or the HMA Manager app can be open-source, *i.e.,* anyone can audit the app and check if it follows the protocols as prescribed. Therefore, assuming that the metadata of the network and the lower communication layers cannot be used to identify users, *e.g.,* by using a proxy or Tor, the HMA App Store cannot link a set of sensitive apps to a user.

10 Evaluation

To evaluate HMA, we used a real dataset of free and paid mHealth apps on the Google Play Store. We looked into three evaluation criteria: (1) overhead experienced by mHealth apps, (2) HMA runtime robustness and its compatibility with mHealth apps, and (3) HMA usability.

10.1 Dataset

We selected 50 apps from the medical category on the Google Play Store, of which 42 apps are free and 8 apps are not. To have a significant and diverse dataset, we selected apps based on their popularity (more than 1000 downloads), their medical specialization, and their supported functionality. From the 50 apps, we filtered out apps that make calls to APIs that we did not support in our prototype implementations, including Google Mobile Services (GMS), Google Cloud Messaging (GCM) and Google Play Services APIs. Note that these services could be supported, similarly to other services, at the cost of additional engineering efforts. We also filtered out apps that use Facebook SDKs, because such SDKs often use custom layouts that are not yet supported by the user-level virtualization lib that HMA uses. Exploring the interaction mechanisms between custom layouts with the Android framework is an avenue for future work.

After filtering, we obtained a set of 30 apps (24 free apps and 6 paid apps) (see the Appendix B) for 15 medical conditions. Also, these apps support features that are crucial for mHealth apps, *e.g.,* a Bluetooth connection with external medical devices (*e.g.,* Beurer HealthManager app [4]) and an internet connection (*e.g.,* Cancer.Net app [5]).

10.2 Implementation Details

Our prototype features the main components of `HMA`, including the `HMA App Store` and the `HMA Manager` app. To measure the operational delay introduced by `HMA`, we implemented a proof-of-concept `HMA App Store` on a computer (Intel Core i7, 3GHz, 16 GB RAM) with MacOS Sierra. Our `HMA App Store` dynamically generated container apps from APKs and relied on an open-source lib called DroidPlugin [8] for user-level virtualization. Our prototype container apps dynamically loaded the apps' classes and resources from the mHealth APKs and supported the interception and proxy of API calls commonly used by mHealth apps, *e.g.,* APIs related to Bluetooth connections and SQLite databases.

10.3 Performance Overhead

In this section, we present the delays introduced by `HMA` to sensitive apps during app-installation and app-launch operations.[31] For the evaluation of delays added by the user-level virtualization to commonly used API methods and system calls at runtime, we refer the readers to existing work, *e.g.,* Boxify [24] shows that such overhead is negligible (opening a camera introduces an overhead of 1.24 ms).

Results presented in this section were measured on a Google Nexus 5X phone running Android 7.0. In our experiments, the `HMA App Store` was connected to the phone through a micro-USB cable, hence network delays were not considered. Yet, compared to the standard use of apps, `HMA` incurs negligible network-delay overheads, because the only bandwidth overhead introduced by `HMA` is the container-app payload whose size is only several hundreds of kilobytes.

10.3.1 App Installation

When a user wants to install an mHealth app, the `HMA App Store` first creates a container app for it. Based on our experiments, assuming the `HMA App Store` decompiles the mHealth APKs beforehand, for 90% of the cases, generating a container app takes, on average, 5 s. Note that a large part of the delay comes from the compilation of the container app, and the measurement was performed on a laptop computer. Also note that the `HMA App Store` can always prepare in advance container apps for each mHealth app, as presented in Section 8.2. The size of the container app is only several hundreds of kilobytes, which takes less than a second for the `HMA Manager` app to download using a 3G or 4G Internet connection. As a result, the total delay overhead introduced by `HMA` would be less than 5 s in the *worst-case* scenario, and less than a second if container apps are generated beforehand, which is acceptable.

10.3.2 App Launch

On Android, apps can be launched from two different states: *cold starts* where apps are launched for the first time since the phone was booted or since the system killed the apps, and *warm starts* where the apps' activities might still reside in memory, and the system only needs to bring them to the foreground, hence faster than cold starts.

Experiment Set-Up. For cold-start delays, we rely on Android's official launch-performance profiling method [13]. For each app, we installed its container app, copied its APK file to its container app's private directory, and launched the container app through `adb`. We then extracted the time information from the `Displayed` entry of the `logcat` output. To simulate a first launch, before we launched an app, we used the command `adb shell pm clear [package-name]` to bring the app back to its initial state. To simulate a cold start, before we launched an app, we used the command `adb shell am force-stop [package-name]` to kill all the foreground activities and background processes of the app. For each app, we collected 50 measurements per launch setting. For a baseline, we measured the delays when the mHealth apps were executed without `HMA`.

To measure warm-start delays, due to the lack of Android supports for profiling warm starts, we have to instrument the source code of the sensitive apps to log the time that the app enters different stages in its lifecycle. Because apps in our dataset are closed source, we used an open-source app.[32] To simulate a warm start, we used the command `input keyevent 187` to bring the app to the background, and then we used the `monkey` command to bring the app back to the foreground. By subtracting the time when the `onResume()` method is successfully executed with the time before the monkey command is sent, we know the warm-start delay experienced by the app. We measured the warm-start delays experienced by the app in both settings (w/ and w/o `HMA`), 50 measurements per setting.

Results. Intuitively, in `HMA`, the first launch of an mHealth will experience longer delays than the subsequent cold starts, because the container app has to process the APK and store the information needed for user-level virtualization. Our experiments show that the median of this process takes 6.5 ± 0.16 s (as compared to 0.74 ± 0.07 s if the mHealth apps were launched w/o `HMA`). Note that this occurs only once, hence it is negligible w.r.t. the lifetime of the app.

Fig. 2 shows the bar plot of subsequent cold-start delays, with and without `HMA`, experienced by mHealth apps; the heights of the bars represent the mean values, and the error bars represent one standard deviation. It can be seen that the average delays are at most 3.0 ± 0.5 s and 1.3 ± 0.05 s if the apps are executed with and without `HMA`, respectively. For 90% of the cases, the average delay with `HMA` is less than 2.0 ± 0.3 s. Note that our prototype is a proof-of-concept hence not optimized. Still, the observed delays are under the

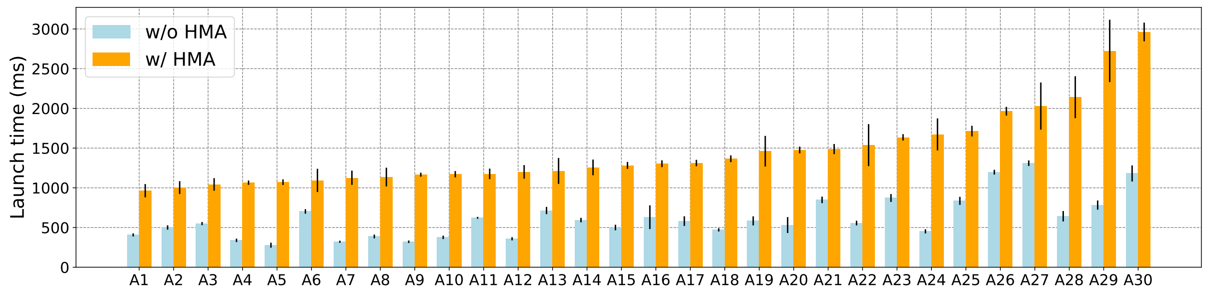

Figure 2: Cold-start delays experienced by mHealth apps when they are executed with and without `HMA`. Note that our `HMA` implementation is a proof-of-concept, hence not-optimized. The heights of the bars represent mean values and the error bars represent the standard deviation. For each setting, we collected 50 measurements per app. The full names of the apps can be found in Table 3 of the Appendices.

delay limit of 5 s suggested by Android [13]. Also, in our user study, 97% of participants agreed that a launch delay of 5 s is acceptable (Section 10.6).

Regarding warm-start delays, we found that the average delay experienced by our tested app, when it was launched with and without `HMA`, was ∼0.55 s. This is intuitive, because the app's processes were still running and the activities still resided in the phone's memory. In case the garbage collector evicts the activities from the phone's memory, warm-start delays can be longer, due to the overheads of activity initializations. We cannot simulate this case, because Android does not provide methods to control the garbage collector. However, in this case, the delay will still be less than cold-start delays (*i.e.,* at most 3 s).

10.4 `HMA` Robustness and Compatibility

In this section, we present the evaluation of `HMA` in terms of its robustness and its compatibility with Android versions.

Runtime Robustness. Following the approach used in previous work, (*e.g.,* [35] and [52]), we manually tested each app in our dataset with `HMA`. For each mHealth app, we extracted its APK, created a container app using `HMA App Store`, and installed the container app on the phone. Thereafter, we used the `HMA Manager` to launch the app. We manually used most of the functionality of the mHealth app, and checked if it had crashed during its execution. We found that all of the apps in our dataset worked normally, except one app that threw an error when making an SQLite connection. To determine the cause of this error, we ran an example app[33] that uses the official Android API for database access (*i.e.,* `Android.database.sqlite`) to insert and retrieve records from an SQLite database, and the example app ran successfully. We suspect that the mHealth app threw an error because it specifies the full path to the database (*i.e.,* `/data/data/package-name/db-name`). Hence, when running the app inside of the `HMA` container app, the hard-coded path is not longer valid. To avoid this problem, developers should specify the relative path to the database (*i.e.,* `./db-name`) instead of its full path.

Compatibility. We ran `HMA` on a series of smartphones with Android OS from version 5.0 to 8.0, which accounts for 89% of the current Android version distribution [7]. We found that `HMA` can be successfully deployed on mainstream commercial Android devices. But, there are two apps (`Mole Mapper` and `Alzheimer's Speed of Processing Game`) that initially failed to run on our Nexus 5X (Android 7.1.1) due to the incompatibility between 32-bit and 64-bit systems. We fixed the problem by enabling the option `-abi armeabi-v7a` when installing them. From the list of 20 apps that we filtered out, we found that 3 apps (`Hearing Aid`, `What's Up` and `Cardiac diagnosis`) successfully ran on Android 5.0 and 6.0, but they failed to run in later versions of Android. We investigated the log of the three apps and found that API methods related to GMS services that we do not support were called in the later versions of Android. This problem could be solved if these services are hooked, as we discussed in Section 10.2. Note that, with the recent release of Android 9, Google has restricted the use of Java reflection[34] – the programming interface that all user-level virtualization techniques rely on. Therefore, for HMA to work seamlessly in Android 9, new user-level virtualization techniques need to be explored. Still, Android 9 has only less than 1% market share [7], which means `HMA` will be compatible with most Android devices in the coming years. Alternatively, HMA could work with rooted Android 9 devices by using dynamic customization frameworks, *e.g.,* Xposed.[35]

10.5 Inter-App Communication Support

Sharing resources with other apps via customized features (*e.g.,* custom permissions and custom intent-filters) and publicly exposing components (*e.g.,* activities, services and content providers) could facilitate fingerprinting attacks. `HMA` can partially support inter-app communications, but it is preferable to avoid such features to guarantee robust fingerprinting protection. Avoiding inter-app communication, however, can affect apps' functionality and backward compatibility. To estimate the effect of using `HMA` and avoiding inter-app communication on existing apps, we analyzed a set of popular sensitive apps from the Google Play Store. Our results show that a small number of apps use inter-app communication features that are not supported by HMA and, in many cases, such features are not directly related to apps' key functionalities.

Dataset. We collected a total of 1045 APK files from the most popular free apps in the Medical and Health&Fitness categories in the US Google Play Store. By checking the apps' descriptions, we found that approximatelly 60% directly match HMA's use case (*i.e.,* health- and fitness-related apps). The rest of the apps are less related to HMA's use case, *e.g.,* apps for medical doctors and nurses, apps for managing accounts with health providers, and apps for managing gym subscriptions. From the APK files, we extracted the manifest file using the `apktool`.

Custom Permissions. Permissions defined by apps to control access to their components can be use to fingerprint them (e.g., they typically include the app name), as any app can list the permissions of other apps. Hence, custom permissions should be avoided. We found that a total of 531 apps declared custom permissions. However, most of these permission declarations are related to deprecated services (Google Cloud Messaging and Android Maps API v2)[36] and can be replaced with newer alternatives that do not require custom permissions. Therefore, ignoring permissions associated with these deprecated services, we found that only 68 apps (6.5%) declared valid custom permissions.

Signature-Level Permissions. Signature permissions[37] are a subset of custom permissions, hence they can be used to fingerprint apps. Given that HMA container apps are all signed by the same key, a malicious app inside a container app could abuse signature permissions to access resources of sensitive apps in other container apps. Therefore, HMA currently does not declare signature permissions in the container apps. Our analysis shows that only 113 apps (10.8%) declared signature permissions and, as explained before, many of these permissions are associated with deprecated services (*e.g.,* Google Cloud Messaging and Android Maps API).

Content Providers. Any app can list the content providers of other installed apps and use this information to fingerprint them. Therefore, HMA obfuscates this information in the container app. This means that public content providers (used to share data with other apps) are not currently supported by HMA. Our analysis shows that only 84 apps (8%) declare public content providers. From these apps, around 68% declare public content-providers associated with third-party frameworks (*e.g.,* Seattle Cloud) for services such as file sharing and authentication, and approximately 23% require permission to access the provider.

Intent Filters. Custom intent filters, *i.e.,* intent filters with app-specific actions, could be used to fingerprint apps. Apps cannot list the intent filters of other apps, but they can list all the activities of other apps that can be performed by a particular intent. Hence, developers should avoid using custom intent filters in their apps' activities. Our analysis shows that this is not a problem, as only 38 apps (3.6%) have activities with custom intent filters.

Activities and Explicit Intents. Explicit intents are currently not supported by HMA because container apps obfuscate the activities' names of sensitive apps; thus, direct reference to sensitive apps' activities is not possible. However, it is recommended to use only explicit intents to launch internal activities; not activities of other apps (implicit intents should be used instead). Our analysis shows that 170 apps (16.2%) declare activities that can be launched by other apps via explicit intents only, *i.e.,* no intent filters. We noticed that many apps (67) declared this type of activities to support Google's Firebase authentication services. Yet, Firebase's official documentation does not seem to mention this approach to support its services. Hence, to be compatible with HMA, these apps could evaluate alternative (official) approaches to support Firebase services or rely on other services for user authentication.

Services and Explicit Intents. Explicit intents are recommended to access services offered by other apps. But, as stated before, it is not possible to use explicit intents with HMA. Our analysis show that 367 apps (35.1%) declare public services that require explicit intents. This is a significant number, yet we noticed that a large number of apps (252) use services with explicit intents to support Google Play services for authentication (Google Sign-In user revocation). Hence, these apps could use alternative user authentication services to be compatible with HMA. We also notice that only 44 apps declared services that belong to the app itself; this indicates that most of these services are associated with third-parties and probably are not part of apps' main functionalities.

Broadcast Receivers. Whereas any app can list the broadcast receivers of other apps, HMA container apps obfuscate the names of the receivers to defend against fingerprinting attacks. The container app can declare the same intent filters for receivers that the sensitive app declares (including custom intent filters) because other apps cannot list these intent filters. As broadcast receivers offer asynchronous communication, nosy apps cannot use API methods to check if an app is receiving a particular broadcast intent. In short, broadcast receivers are supported by HMA.

10.6 HMA Usability and Desirability

To evaluate the usability of HMA and the users' interest for it, we conducted a user study that was approved by our institutional ethical committee. It involved 30 student subjects (19 males, 11 females, 22 ± 4.5 years old) from 18 areas of study. The participants were experienced Android users: 87% of them have used an Android phone for at least a year. Also, they were relatively concerned about their privacy; using the standard metric for measuring privacy perception (IUIPC) [39], we found that, on a scale from 1 to 5, 97% of participants graded at least 3.0 and an average of 4.1.

We began the study with an entry survey about demographic information, privacy postures, users' awareness and concerns about the problem of `LIA` collections. Then, we provided each participant with a fresh phone and asked them to install and use two apps: a popular public-transportation app for our city and an mHealth app called `Cancer.Net`. To precisely measure the users' perceptions of the delay introduced by `HMA`, the participants were asked to use the two aforementioned apps with and without `HMA`; detailed instructions were provided to them. 67% of the participants had used the transportation app before, whereas only 7% of them had used the `Cancer.Net` app or an mHealth app. We finished the user study with an exit survey containing questions related to the usability of `HMA` and the users' levels of interest in `HMA`. The user-study session took ~45 minutes, and we paid each participant ~$25 (*i.e.,* 25 CHF). The transcript of survey questions and the instructions can be found at [38].

Our study shows that the participants are concerned about the privacy of health-related data: 90% of the participants would be at least concerned if their health-related information were collected by apps installed on their phones and shared with third parties, and 87% of participants would be at least concerned if third parties learned that they had used health-related apps. Indeed, our study confirms the findings from previous works (*e.g.,* [41]) that the majority of people never read privacy policies. Therefore, the current solution of using privacy policies by Google for `LIA` collections is not satisfactory. These findings make clear the case for `HMA`.

Regarding the usability of `HMA`, only 30% of the participants noticed a difference when the two apps ran with and without `HMA`. Note that the delays that users experienced in the user study were the first-launch delays, which are 4.2 ± 0.06 s and 5.1 ± 0.07 s for the transportation app and the `Cancer.Net` app, respectively. From the open-ended question in our exit survey, we found that the observed differences are mainly about the launching delay of the apps and the change in the app names in permission prompts. From the close-ended questions, which were coded using a five-point Likert scale, we observe the following. Almost all participants agree that these changes and delays are acceptable (97% and 93% of the participants, respectively). 93% of the participants also agree that the use of an `HMA Manager` to install and launch apps is at least somewhat acceptable. Also, 90% of the participants agreed that `HMA` does not affect the user experience of the apps that it protects, and that they are at least somewhat interested in using `HMA`. These results suggest that `HMA` is usable and desirable.

11 Conclusion

In this work, we have shown that apps can collect a significant amount of static and runtime information about other apps, to fingerprint them. Our analysis has shown that many third parties are interested in learning about the apps installed on people's phones. Moreover, we have shown that there are no existing mechanisms for hiding the presence of an app from other apps. We have proposed `HMA`, the first solution that addresses this problem. `HMA` does not require any modifications to the Android OS and preserves the key functionalities of apps. The results of our evaluation and user study suggest that `HMA` is usable and of interest to users.

References

[1] Android Security 2015 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf. Visited: Sep. 2018.

[2] Android Security 2016 Year In Review. https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf. Visited: Sep. 2018.

[3] Angry Birds. https://play.google.com/store/apps/details?id=com.rovio.angrybirds. Visited: Sep. 2018.

[4] Beurer HealthManager. https://play.google.com/store/apps/details?id=com.beurer.connect.healthmanager. Visited: Sep. 2018.

[5] Cancer.Net Mobile. https://play.google.com/store/apps/details?id=com.fueled.cancernet. Visited: Sep. 2018.

[6] DH Texas Poker - Texas Hold'em. https://play.google.com/store/apps/details?id=com.droidhen.game.poker. Visited: Sep. 2018.

[7] Distribution dashboard. https://developer.android.com/about/dashboards/. Visited: Sep. 2018.

[8] DroidPlugin. https://github.com/DroidPluginTeam/DroidPlugin. Visited: Sep. 2018.

[9] F-Droid. https://f-droid.org/en/. Visited: Sep. 2018.

[10] Help protect against harmful apps with Google Play Protect. https://support.google.com/accounts/answer/2812853?hl=en. Visited: Sep. 2018.

[11] Hide App, Private Dating, Safe Chat - PrivacyHider. https://play.google.com/store/apps/details?id=com.trigtech.privateme&hl=en. Visited: Sep. 2018.

[12] InstaSize Editor: Photo Filters and Collage Maker. https://play.google.com/store/apps/details?id=com.jsdev.instasize. Visited: Sep. 2018.

[13] Launch-Time Performance. https://developer.android.com/topic/performance/launch-time.html. Visited: Sep. 2018.

[14] MX Player. https://play.google.com/store/apps/details?id=com.mxtech.videoplayer.ad. Visited: Sep. 2018.

[15] Neon Motocross. https://play.google.com/store/apps/details?id=com.motomex.neonmotocross. Visited: Sep. 2018.

[16] Nova Launcher. https://play.google.com/store/apps/details?id=com.teslacoilsw.launcher&hl=en. Visited: Sep. 2018.

[17] Parallel Space - Multiple accounts and Two face. https://play.google.com/store/apps/details?id=com.lbe.parallel.intl&hl=en. Visited: Sep. 2018.

[18] Private Zone - Safe Vault. https://play.google.com/store/apps/details?id=com.leo.appmaster. Visited: Sep. 2018.

[19] Publish Your App. https://developer.android.com/studio/publish/index.html#publishing-unknown. Visited: Sep. 2018.

[20] Solitaire: Super Challenges. https://play.google.com/store/apps/details?id=com.cardgame.solitaire.full. Visited: Sep. 2018.

[21] Sweet Selfie - selfie camera, beauty cam, photo edit. https://play.google.com/store/apps/details?id=com.cam001.selfie. Visited: Sep. 2018.

[22] Achara, J. P., Acs, G., and Castelluccia, C. On the Unicity of Smartphone Applications. In *Proc. of WPES* (2015).

[23] Aitken, M., and Lyle, J. Patient adoption of mhealth: use, evidence and remaining barriers to mainstream acceptance. *IMS Institute for Healthcare Informatics* (2015).

[24] Backes, M., Bugiel, S., Hammer, C., Schranz, O., and von Styp-Rekowsky, P. Boxify: Full-fledged App Sandboxing for Stock Android. In *Proc. of USENIX Security* (2015).

[25] Bianchi, A., Fratantonio, Y., Kruegel, C., and Vigna, G. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proc. of SPSM* (2015).

[26] Chen, Q. A., Qian, Z., and Mao, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of USENIX Security* (2014).

[27] Chen, Y., Jin, X., Sun, J., Zhang, R., and Zhang, Y. POWERFUL: Mobile app fingerprinting via power analysis. In *Proc. of IEEE INFOCOM* (2017).

[28] Dai, S., Tongaonkar, A., Wang, X., Nucci, A., and Song, D. NetworkProfiler: Towards automatic fingerprinting of Android apps. In *Proc. of IEEE INFOCOM* (2013).

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

[30] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. of SOUPS* (2012).

[31] Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., and Prakash, A. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proc. of USENIX Security* (2016).

[32] Grace, M. C., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. of ACM WiSec* (2012).

[33] Gulyás, G. G., Acs, G., and Castelluccia, C. Near-Optimal Fingerprinting with Constraints. *Proceedings of Privacy Enhancing Technologies Symposium* (2016).

[34] Huang, J., Schranz, O., Bugiel, S., and Backes, M. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *Proc. of ACM CCS* (2017).

[35] Jaebaek, S., Daehyeok, K., Donghyun, C., Insik, S., and Taesoo, K. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Proc. of NDSS* (2016).

[36] Jana, S., and Shmatikov, V. Memento: Learning secrets from process footprints. In *Proc. of IEEE S&P* (2012).

[37] Kotz, D., Gunter, C. A., Kumar, S., and Weiner, J. P. Privacy and Security in Mobile Health: A Research Agenda. *Computer* (June 2016).

[38] Lin, C.-C., Li, H., Zhou, X.-y., and Wang, X. Screenmilker: How to Milk Your Android Screen for Secrets. In *Proc. of NDSS* (2014).

[39] Malhotra, N. K., Kim, S. S., and Agarwal, J. Internet users' information privacy concerns (IUIPC): The construct, the scale, and a causal model. *Information systems research* (2004).

[40] Malmi, E., and Weber, I. You Are What Apps You Use: Demographic Prediction Based on User's Apps. In *Proc. of AAAI CWSM* (2016).

[41] McDonald, A. M., Reeder, R. W., Kelley, P. G., and Cranor, L. F. A Comparative Study of Online Privacy Policies and Formats. In *Privacy Enhancing Technologies* (2009).

[42] Naveed, M., Zhou, X.-y., Demetriou, S., Wang, X., and Gunter, C. A. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android. In *Proc. of NDSS* (2014).

[43] Nguyen, D. C., Wermke, D., Acar, Y., Backes, M., Weir, C., and Fahl, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proc. of ACM CCS* (2017).

[44] Ratazzi, P., Aafer, Y., Ahlawat, A., Hao, H., Wang, Y., and Du, W. A systematic security evaluation of android's multi-user framework. *arXiv preprint arXiv:1410.7752* (2014).

[45] Seneviratne, S., Seneviratne, A., Mohapatra, P., and Mahanti, A. Predicting User Traits from a Snapshot of Apps Installed on a Smartphone. *SIGMOBILE Mob. Comput. Commun. Rev. 18*, 2 (June 2014).

[46] Seneviratne, S., Seneviratne, A., Mohapatra, P., and Mahanti, A. Your installed apps reveal your gender and more! *SIGMOBILE Mob. Comput. Commun. Rev.* (2015).

[47] Sun, M., and Tan, G. NativeGuard: Protecting Android Applications from Third-party Native Libraries. In *Proc. of ACM WiSec* (2014).

[48] Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N., and Cranor, L. F. Crying Wolf: An Empirical Study of SSL Warning Effectiveness. In *Proc. of USENIX Security* (2009).

[49] Taylor, V. F., Spolaor, R., Conti, M., and Martinovic, I. Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic. In *Proc. of IEEE EuroS&P* (2016).

[50] Taylor, V. F., Spolaor, R., Conti, M., and Martinovic, I. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Trans. on Inf. Forensics and Security 13*, 1 (Jan. 2018).

[51] Wang, T., and Goldberg, I. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *Proc. of USENIX Security* (2017).

[52] Wang, X., Sun, K., Wang, Y., and Jing, J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *Proc. of NDSS* (2015).

[53] Xu, Q., Liao, Y., Miskovic, S., Mao, Z. M., Baldi, M., Nucci, A., and Andrews, T. Automatic generation of mobile app signatures from traffic observations. In *Proc. of IEEE INFOCOM* (2015).

[54] Zhou, X., Demetriou, S., He, D., Naveed, M., Pan, X., Wang, X., Gunter, C. A., and Nahrstedt, K. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proc. of ACM CCS* (2013).

Notes

[1] https://liquid-state.com/mhealth-apps-market-snapshot/. Visited: Nov. 2018.

[2] https://research2guidance.com/mhealth-app-market-getting-crowded-259000-mhealth-apps-now/. Visited: Sep. 2018.

[3] https://www.theguardian.com/technology/2014/nov/27/twitter-scanning-other-apps-tailored-content. Note that Twitter recently announced that it excludes apps dealing with health, religion and sexual orientation, https://help.twitter.com/en/safety-and-security/app-graph. Visited: Sep. 2018.

[4] https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/. Visited: Sep. 2018.

[5] Now reclassified as Mobile Unwanted Software (MUwS) [2].

[6] https://play.google.com/about/developer-content-policy-print/. Visited: Sep. 2018.

[7] Additional protections by Safe Browsing for Android users, https://security.googleblog.com/2017/12/additional-protections-by-safe-browsing.html. Visited: Sep. 2018.

[8] Note that, unlike previous work (*e.g.,* [22]) that focuses on apps directly retrieving the list of installed apps, our work focuses on the fingerprintability of *a specific* app, a more general and difficult problem.

[9] https://developer.android.com/guide/topics/permissions/overview. Visited: Sep. 2018.

[10] https://developer.android.com/studio/command-line/adb.html. Visited: Sep. 2018.

[11] https://codelabs.developers.google.com/codelabs/developing-android-a11y-service/. Visited: Apr. 2019.

[12] https://cromulentlabs.wordpress.com/2016/01/15/explanation-of-canopenurl-changes-in-ios-9/. Visited: Sep. 2018.

[13] https://ibotpeaches.github.io/Apktool/. Visited: Sep. 2018.

[14] Note that we also found many occurrences of other methods presented in Section 4, but we did not know the purposes of the calling apps.

[15] http://www.zdnet.com/article/accuweather-caught-sending-geo-location-data-even-when-denied-access/. Visited: Nov. 2018.

[16] https://github.com/M66B/XPrivacy. Visited: Sep. 2018.

[17] We performed a similar analysis on a small set of paid apps, see Appendix A.

[18] https://play.google.com/about/privacy-security-deception/personal-sensitive/. Visited: Sep. 2018.

[19] https://source.android.com/devices/tech/admin/multi-user. Visited: Sep. 2018.

[20] https://www.xda-developers.com/add-multi-user-support-android/. Visited: Feb. 2019.

[21] https://www.android.com/enterprise/employees/. Visited: Sep. 2018

[22] https://developer.android.com/topic/instant-apps/index.html. Visited: Sep. 2018.

[23] https://developer.android.com/topic/instant-apps/reference.html#instantapps.InstantApps. Visited: Sep. 2018

[24] https://www.samsungknox.com/en. Visited: Sep. 2018.

[25] https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011. Visited: Sep. 2018.

[26] P is defined based on the estimation about the number of sensitive apps that users of the `HMA App Store` can have, because Android does not permit duplicate package names for apps. Average users have around 80 apps on their phones, therefore P is *at most* 80.

[27] https://developer.android.com/about/versions/marshmallow/android-6.0#direct-share. Visited: Feb. 2019.

[28] https://www.torproject.org/projects/torbrowser/design/. Visited: Sep. 2018.

[29] https://www.androidpolice.com/2017/11/12/google-will-remove-play-store-apps-use-accessibility-services-anything-except-helping-disabled-users/. Visited: Apr. 2018.

[30] A signature-protected permission is a permission that the system grants only if the requesting app is signed with the same certificate as the app that declared the permission.

[31] We omit the app-update operation, because app-update and app-installation operations are similar.

[32] https://github.com/commonsguy/cw-omnibus/tree/master/Activities/Lifecycle. Visited: Sep. 2018.

[33] SQLiteOpenHelper, https://github.com/commonsguy/cw-omnibus/tree/master/Database/ConstantsROWID. Visited: Sep. 2018

[34] https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces. Visited: Sep. 2018.

[35] https://repo.xposed.info/module/de.robv.android.xposed.installer. Visited: Sep. 2018

[36] https://developers.google.com/cloud-messaging/android/android-migrate-fcm. Visited: Feb. 2019.

[37] A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission

[38] https://www.dropbox.com/sh/lo273jtx6jkbf1c/AAB1BtkBmBuNVOV13OAwDu-ha?dl=1

A Analysis of Paid Apps

To estimate if there are differences between free and paid apps w.r.t. collecting `LIAs`, we performed a similar analysis with a set of 28 popular paid apps from different categories in the Google Play Store. We found that 17.8% of the paid apps included at least one call to `getIA()` or `getIP()` methods in their code (*upper-bound*) and that 7.4% of the paid apps called at least one of these two methods at runtime (*lower-bound*). Although the number of paid apps evaluated is much smaller than of free apps, our results still indicate that paid apps are less likely to query for `LIAs`, probably because they rely less on third-party libs, particularly ad libraries.

B Apps Tested with `HMA`

Index	App Name	# Downloads
1	My Ovulation Calculator	1M - 5M
2	Blood Pressure Log - MyDiary	500K - 1M
3	DreamMapper	100K - 500K
4	Breathing Zone	5K - 10K
5	Alzheimer's Speed of Processing Game	1K - 5K
6	Cancer.Net Mobile	10K - 50K
7	AIDS Info Drug Database	5K - 10K
8	My Pain Diary	5K - 10K
9	iBP Blood Pressure	10K - 50K
10	Squeezy: NHS Pelvic Floor App	10K - 50K
11	OneTouch Reveal	500K - 1M
12	ADHD Adults	10K - 50K
13	Baritastic - Bariatric Tracker	100K - 500K
14	Mole Mapper	1K - 5K
15	Respiroguide Pro	10K - 50K
16	FearTools - Anxiety Aid	10K - 50K
17	Back Pain Relieving Exercises	10K - 50K
18	OnTrack Diabetes	500K - 1M
19	Asthmatic	50 - 100
20	MoodTools - Depression Aid	100K - 500K
21	Pain Diary & Forum CatchMyPain	50K - 100K
22	Beurer HealthManager	100K - 500K
23	Constant Therapy	10K - 50K
24	Self-help Anxiety Management	100K - 500K
25	Pregnancy Calendar and Tracker	1M - 5M
26	BELONG Beating Cancer Together	10K - 50K
27	AsthmaMD	10K - 50K
28	Propeller	5K - 10K
29	mySugr: the blood sugar tracker made just for you	500K - 1M
30	QuitNow! - Quit smoking	1M - 5M

Table 3: mHealth apps tested with `HMA`.