Hiding the Presence of Sensitive Apps on Android

Anh Pham[1], Italo Dacosta[1], Eleonora Losiouk[2], John Stephan[1]
Kévin Huguenin[3], Jean-Pierre Hubaux[1]
[1]School of Computer and Communication Sciences (IC), EPFL, Lausanne, Switzerland
[2]University of Padova
[3]Faculty of Business and Economics (HEC), UNIL, Lausanne, 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 (*e.g.,* diabetes app) is installed on a users' phone, can reveal sensitive information about the user's health. Android's open design enables apps without any permissions to easily check for the presence of a specific app or to collect the entire list of installed apps on the phone, even if multiple user accounts or profiles are used. To date, no mechanisms exist to effectively hide the use of sensitive apps. Our analysis of 2917 popular apps in the Google Play Store shows that around 57% of these apps explicitly query for the list of installed apps. We also show that Android apps expose a significant amount of identifiable metadata that can be used for fingerprinting attacks. Therefore, we designed and implemented `HideMyApp` (`HMA`), a practical and robust solution that hides the presence of mHealth and other sensitive apps from nosy apps on the same phone. `HMA` relies on user-level virtualization techniques, thus avoiding changes to the operating system (OS) or to apps while still supporting key functionality. By using a diverse dataset of both free and paid mHealth apps, our experimental evaluation shows that `HMA` supports main features in most apps and introduces acceptable delays at runtime, *i.e.,* several milliseconds; such 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`, can provide robust protection against such a threat.**

I. Introduction

Mobile health (mHealth), the use of technologies such as smartphones, mobile apps and wearable sensors for wellness and medical purposes, promises to improve the quality 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. For instance, more than a third of physicians in the US recommend mHealth apps to their patients [1], and there are around 325,000 mHealth apps available in major app stores [2].

Given the sensitivity of medical data, privacy and security are one of the main challenges to the success of mHealth technologies [3]. In this area, a serious and often overlooked threat is that an adversary can infer sensitive information from the presence of apps 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 [4]–[7]. With the increasing popularity of mHealth apps, an adversary can now infer much more sensitive information. For example, learning that a user has apps to manage diabetes and depression can be very telling about the user's medical conditions; such information could be misused to profile, discriminate or blackmail users. When inquired about it, 87% of the participants in our user-study expressed concern about this threat (Section X-E).

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 through standard APIs [8]. Our analysis shows that Android exposes a considerable amount of static and runtime metadata about installed apps (Section IV); this information can be misused by nosy apps to accurately fingerprint other apps installed on the phone. Such attacks are possible even when the sensitive apps and the nosy app are installed on different user accounts or profiles on a single phone. In 2014, Twitter was criticized for collecting the list of installed apps to offer targeted ads [9].[1] 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 around 57% of these apps include calls to API methods that explicitly collect the list of installed apps (Section V). Our analysis also shows that free apps are more likely to query for such information and that the use of third-party libraries (libs), *e.g.,* ad libs, is the main reason for such behavior, corroborating the findings of previous studies [6], [11]. As users can have on average around 80 apps installed on their phones [12], most of them free, there is a high chance of untrusted third-parties obtaining the list of installed apps.

Since 2015, Android started to classify as potentially harmful apps (PHA)[2] the apps that collect information about installed apps without user consent [14]. To avoid such classification, however, developers simply need to provide a privacy policy describing how the app collects, uses and shares sensitive user data [15]. Interestingly, we found that only 7.7% of the evaluated apps declared that they collect the list of installed apps in their privacy policies; several of these policies use similar generic statements (probably based on a common template), and some state that such a list is non-personal information (Section V-D). Moreover, few users read and understand privacy policies [16], as our user-study also

[1]Twitter recently announced that it excludes apps dealing with health, religion and sexual orientation [10].

[2]Now reclassified as mobile unwanted software (MUwS) [13].

confirmed (Section X-E). Android recently announced that Android's security services will display warnings on apps that collect without consent users' personal information, including the list of installed apps [17]. This is a welcomed step, but the effectiveness of security warnings is known to be limited [18], [19] and it is unclear how queries by third-party libs will be handled. It is also unclear if such mechanisms will be able to detect more subtle attacks, where a nosy app checks for the existence of a small set of sensitive apps using more advanced fingerprinting techniques (Section IV). In addition, Android does not provide mechanisms to hide sensitive apps installed on their phones; a few third-party tools, designed for other purposes, can provide only partial protection to some users (Section VI). *Therefore, it is crucial to have a solution that effectively hides the presence of sensitive apps from other apps in the same phone.*

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 app has all Android permissions and debugging privilege (ADB). 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 `HMA` manager app to (un)install, use and update the apps selected from the `HMA` app 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 OS. At the same time, `HMA` preserves the key functionality of mHealth apps, such as connecting to external devices via Bluetooth, sending information to online services over the Internet and storing information in a database. Moreover, `HMA` introduces only negligible effects on the usability of the apps.

`HMA` enables users to launch a sensitive app inside the context of a container app, without requiring the sensitive app to be installed, *i.e.,* the sensitive app will not be registered in the OS. A container app is a dynamically generated wrapper around the APK (Android application package) of the sensitive app, and it is designed in such a way that the sensitive app cannot be fingerprinted. In order 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.,* [20], [21]. To reduce the amount of identifiable information exposed by the app, we provide developers with guidelines on how to reduce the identifiable metadata in the app and, at runtime, the container app dynamically intercepts and obfuscates identifiable information output by the protected app. Note that, the main contributions of our work is not about a new user-level virtualization technique, but rather the analysis of possible information leaks that an nosy app can exploit to fingerprint sensitive apps, and a practical design and evaluation of a mechanism to effectively reduce such leaks.

Our thorough evaluation of `HMA` on a diverse data-set of both free and paid mHealth apps on the Google Play store shows that `HMA` is practical and that it introduces reasonable operational delay to the users. For example, in 90% of the cases, the delay introduced by `HMA` to the cold start of an mHealth app (*i.e.,* the app is launched for the first time since the device booted, or since the system terminated it) by a non-optimized proof-of-concept implementation of `HMA` is less than 1 second. At runtime, the delay introduced is in the order of tenths of milliseconds and is unlikely to be noticed by users. Moreover, our user-study, involving 30 participants, suggests that `HMA` is user-friendly and of interest to users.

In short, we are the first to systematically investigate this important yet understudied problem, and to propose a practical solution. Our main contributions are as follows.

- *A new perspective*: We identify a crucial and novel problem: the fact that nosy apps can easily fingerprint and identify other installed apps is particularly severe, especially in mHealth, where the presence of an app can reveal user's medical conditions.
- *Systemized knowledge*: We are the first to systematically investigate the different techniques that a nosy app can use to fingerprint another app in the same phone. Also, through our thorough static and dynamic analysis of paid and free apps across all app categories in the Google Play store, we gain insights into the prevalence of the problem.
- *Design and implementation of a solution for hiding sensitive apps*: We present `HideMyApp`, a practical system that provides robust defense against fingerprinting attacks targeting sensitive apps in Android. `HideMyApp` works on stock Android, and no firmware modification or root privilege is required.
- *Thorough evaluation of our solution*: The evaluation of `HMA`'s prototype on apps from the Google Play store suggests that our solution is practical and usable. Moreover, the results of our user study suggest that `HMA` is perceived as usable and of interest to the users. `HMA`'s source code and its experimental results are available online at [22].

II. 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 [23], [24], and that apps have suspicious activities, *e.g.,* they collect call logs, phone numbers and browser bookmarks [6], [11], [25]. Zhou *et al.* [8] show that Android's open design has made publicly available a number of seemingly innocuous phone's resources, including the list of installed apps, that could be used to infer sensitive information about their users, such as users' gender and religion [4]–[7], [26]. Similarly, Chen *et al.* present a way to fingerprint Android apps based on their power consumption profiles. Moreover, a significant amount of work has been devoted to fingerprinting Android apps based on their (encrypted) network traffic patterns [27]–[30]. Researchers have also shown that it is possible to perform re-identification attacks based on

a small subset of installed apps [7], [31]. Closer to our work, Demetriou *et al.* [6] performed a static analysis to quantify the prevalence of the collections of the list of installed apps and their meta-data by third-party libs. We go beyond their work, however, by complementing the limitations of static analyses by performing a dynamic analysis and privacy-policy analysis and designing a solution to address the problem.

Existing mechanisms for preventing nosy apps from learning about the presence of another app are not sufficient (Section VI). As we will show in Section VIII, 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.,* [21], [32]) or to compartmentalize third-party libs from the host apps (*e.g.,* [33]). However, as they were designed for a different problem, they do not directly help to hide the presence of a sensitive target app on a phone. They either require the target apps to be first installed on the phone, thus exposing them to nosy apps through public APIs (Section IV), or they grant the container apps with all possible Android permissions to enable multiple target apps to run inside the same container app, thus violating the Android's app-isolation and least-privilege security models (Section VII-C).

III. Background

In this section, we provide background knowledge on Android apps and Android security models.

A. Android Security Model

Android requires each app to have a unique package name that is defined by its app 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 that only this UID has read and write permissions. Additionally, each app is executed in a dedicated process. Thus, apps are isolated, or sandboxed, both at the process and file levels. Apps interact with the underlying system (*i.e.,* the app framework and OS) via methods defined by the Java API framework and the Linux-layer interfaces. Depending on the API methods and commands, apps might need to be granted certain permissions [34] or privileges (*e.g.,* [35]). Beginning with Android 6.0, users can grant permissions to apps at runtime.

B. Android Apps and APK Files

Android apps are usually written in Java, then compiled into Dalvik bytecode (.dex format) to be executed on Android devices. Each app must contain a set of *mandatory* information, including a unique package name, an icon and label, a folder containing the app's resources, and at least one of the following components: Activity, Service, Broadcast Receiver and Content Provider [36]. Apps can also support *optional* features, including implicit intents, permissions, shared libraries, customized app configurations (*e.g.,* supported SDKs, themes and styles), custom permissions, and assets.

Apps are distributed in the form of APK files, and they must be digitally signed by their developers. An APK is a zip archive that contains the compiled source code and resources of the app. Each APK also includes a manifest configuration file, called `AndroidManifest.xml`, that contains a description of the app, *e.g.,* its package name and components.

IV. Fingerprintability of Android Apps

In this section, we show that Android apps disclose a significant amount of information that can be used by nosy apps to perform fingerprinting attacks. This information includes *static information* (*i.e.,* information available after apps are installed and that typically does not change during apps' lifetime), and *runtime information* (*i.e.,* information generated or updated by apps at runtime). To be exhaustive, our analysis, conducted on Android 8.0, focuses on information leaks through both Android's Java API framework and its Linux-layer interface, w.r.t. the granted permissions and privileges of the nosy app. Our findings are summarized in Table I and detailed below.

A. Information Leaks Through the Java API Framework

Static Information. *Without permissions*[3], a nosy app can directly check if a specific package name exists on the phone (a package name is mandatory and unique for each app). This can be done by invoking two methods `getInstalledApplications()` and `getInstalledPackages()`. These methods return the entire list of installed package names. Moreover, apps can register broadcast receivers, such as `PACKAGE_INSTALLED` and `PACKAGE_ADDED`, to be notified when an app is installed. Apps can also use various methods of the `PackageManager` class, *e.g.,* `getResourcesForApplication()` as an oracle to check for the presence of an app. These methods take a package name as a parameter and return *null* if the app does not exist.

If Android restricts access to apps' package names, i.e., by requiring permission(s), a nosy app can still retrieve other static information for fingerprinting attacks. For example, apps' mandatory information (*e.g.,* components' names, icons, labels, developers' signatures and signing certificates), and optional features (*e.g.,* implicit intents, requested permissions, shared libraries, custom permissions, assets and apps' configurations such as supported SDKs, themes and styles). This information can be obtained through a number of methods in the `PackageManager` class, *e.g.,* `getPackageInfo()`.

Runtime information. *Without permissions*, apps cannot obtain runtime information generated by other apps. However, *with permissions*, apps can obtain some identifying information. For instance, the `PACKAGE_USAGE_STATS` permission allows an app to obtain the list of running processes (using `getRunningAppProcesses()`), and statistics about network and storage consumption of all installed apps, including their package names, during a time interval (using `queryUsageStats()`). Apps granted with

[3]Note: Granting permissions to a zero-permission app does not enable it to collect additional static information about other apps.

	Java API Framework		Linux-Layer Interface	
	W/o Permissions	**W/ Permissions**	**Default App Privilege**	**ADB Privilege**
Static Information	*Mandatory information:* + Package name + Components' names *Optional features:* + Implicit intent filters + Requested permissions	(*) See note	+ Package names	+ Package names + APK path + APK file
Runtime Information	None	+ Files in external storage + Storage consumption + Running processes + Network traffic + Package names + Notifications	+ UI states† + Power consumption† + Memory footprints†	+ Files in external storage + Network consumption + Running processes + Screenshots + System log + System diagnostic outputs

TABLE I: Identifying information about installed apps that a nosy 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, † means that the information could be learnt in older versions of Android, but later versions (*e.g.,* Android 8.0) require the calling app to have adb privilege. **(*) Note:** Granting permissions to a zero-permission nosy app does not enable it to obtain more static information about apps.

the `READ_EXTERNAL_STORAGE` permission, a frequently requested permission, can inspect for unique folders and files in the phone's external storage (a.k.a SD card). Also, apps with the `BIND_NOTIFICATION_LISTENER_SERVICE` permission can receive notifications sent to other apps. Moreover, apps with VPN capabilities (permission `BIND_VPN_SERVICE`) can intercept network traffic of other apps; Existing work shows that network traffic can be used to fingerprint apps with good accuracy [28], [30], [37].

B. Information Leaks Through the Linux-Layer Interface

Static Information. With *default app privilege*, apps 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. With debugging privilege (*i.e., adb privilege*), an app can retrieve the list of package names (*e.g.,* using the command`pm list packages`) and learn the path to the APK file of a specific app (*e.g.,* using the command `pm path [package name]`). Moreover, adb privilege enable a nosy app to retrieve the APK files of other apps (*e.g.,* using the command `pull [APK path]`); The nosy app can then use a method such as `getPackageArchiveInfo()` to extra identifying information from the APK files.

Runtime Information. With *default app privilege*, apps can infer the UI states [38], memory footprints [39] and power consumption [40] of other installed apps. Note that access to this information has been restricted in later versions of Android (*e.g.,* Android 8.0 requires the apps to have adb privileges). Additionally, with *adb privilege*, apps can learn detailed information about runtime behaviors of other apps by inspecting the system logs and diagnostic outputs (using commands such as `logcat` and `dumpsys`, respectively). Moreover, with adb privileges, apps can directly retrieve the list of running processes (*e.g.,* using command `ps`), take screenshots [41] or gain access to statistics about network usage of other apps (folder `/proc/uid_stat/[uid]`).

Our analysis shows that Android's open design makes it easy for a nosy app to fingerprint other apps on the same phone. The diverse source of identifying information leaks makes it difficult to add permissions or block all related methods, particularly because most of these methods have valid use cases and are widely used by apps; restricting or blocking them could break a significant number of apps and anger developers. We are the first to provide a solution to hide the presence of sensitive apps w.r.t. a strong adversary that has access to all the aforementioned information (Section VII).

V. Apps Inquiring About Other Apps

We analyze popular apps from the Google Play store to estimate how common it is for apps to inquiry about other apps on the same phone. Our analysis focuses on apps that contain API calls to directly retrieve the list of installed apps (hereafter `LIA`): `getInstalledApplications()` and `getInstalledPackages()` (hereafter abbreviated as `getIA()` and `getIP()`, respectively). We chose these methods because their use clearly indicates the intent of apps to learn about the presence of other apps, whereas other methods presented in Section IV, such as `getPackageInfo()`, can be used in valid use cases or for app fingerprinting attacks. Therefore, the results presented in this section can be interpreted as a *lower-bound* on the number of apps that are interested in identifying other installed apps.

A. 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, *i.e.,* top 60 apps. After eliminating duplicate entries (an app can belong to multiple categories), apps already installed on Android by default, and brand-specific apps, we were left with 2917 distinct 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' Play store pages, *i.e.,* 418 apps did not have privacy policies. Note that Google requires developers to post a privacy policy in both the designated field in the app's

Analysis method	Call origin	`getIA()` (%)	`getIP()` (%)	`getIA()` or `getIP()` (%)
Static	Third-party libs + Apps	36.4	43.6	57.0
Static	Apps only	8.05	8.43	13.9
Dynamic	Third-party libs + Apps	6.54	15.0	19.2

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

Play store page and within the app itself, if their apps handle personal or sensitive user-data [42].

B. Static Analysis

For our static analysis, we decompiled the APKs in our dataset to obtain their smali code, a human-readable representation of the app's bytecode, by using the well-known tool Apktool [43]. From the smali code, we searched for occurrences of two methods `getIA()` and `getIP()`.[4] API calls can be located in three parts of the decompiled code: in code of Android/Google libs and SDKs (e.g., Android AppCompat supporting library), in code of third-party libs and SDKs (e.g., Flurry analytics, Facebook SDK), or in code of the app itself. To differentiate among these three origins, we applied the following heuristics. 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. This is a reasonable heuristic, because Android follows the Java package name conventions with the reversed Internet domain of the companies, generally has length of two words, to begin their package names. If the methods do not match the first two categories, then they are considered part of the code of a third-party library or SDK. Note that this approach, also used in previous work [6], cannot precisely classify obfuscated code or code in paths with no meaningful names (*e.g.,* developers might not follow naming conventions). Such cases, which represent a small fraction (*i.e.,* less than 5%), are classified as code from third-party libs or SDKs. Still, this approach provides us with a good estimation of the origin of most method calls.

Table II shows the proportions of apps that invoke two sensitive methods *i.e.,* `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 sensitive 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.* [6]. These results also show that most sensitive requests are originated from the code belonging to third-party libs or SDKs; app developers might not be aware of this activity, as it has been the case for other sensitive data such as location [44].

Static analysis has two main limitations. First, it might be the case that methods appeared in the code are never executed by the app, *i.e.,* a false positive. Second, it is possible that the sensitive methods do not appear in the code included in the APK, rather in the code loaded dynamically by the app or the third-party libs at runtime, *e.g.,* by using Java reflection [45]. To address these issues, we also performed a dynamic analysis of the apps in our dataset, as we describe next.

C. Dynamic Analysis

For our dynamic analysis, we intercepted the API calls made by apps by using XPrivacy [46] on a phone with Android 6.0. To scale the analysis, for each app, we installed the app and granted it all the permissions requested (declared in the app's Manifest file). Next, we launched all the runnable activities declared by the app in an interval of 10 minutes, trying to approximate the actions a user might trigger while using the app. Although this approach has limitations, *e.g.,* short period of time per app and it cannot emulate all the activities a user could do, it is enough to estimate a *lower-bound* on the number of apps that query for `LIAs` at runtime.

Our results, presented in Table II, show that, 190 (6.54%) apps called `getIA()` and 436 (15.0%) apps called `getIP()`. This means that 19.2% of the apps in our dataset called at least one of these two sensitive methods during the first 10 minutes after being installed and executed. However, because it is not possible to infer the origin of the request (*i.e.,* Google/Android library, third-party library, or the app itself) from the data gathered, we performed some additional steps. For each app, we used the results of our static analysis (Section V-B) 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()`. This means that, with good certainty, the majority of these sensitive requests originated from third-party libs or app codes.

Interestingly, we found 49 apps that called at least one of the two 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 sensitive requests were dynamically loaded at runtime. Still, this represents a small percentage of the apps found by our analysis.

Our static and dynamic analysis have shown that a significant number of popular free apps in the Google Play store actively queries for `LIAs`: between 19.2% (dynamic analysis) and 57% (static analysis) of the apps in our dataset. In addition, our results show that third-party libs are probably mainly responsible for such sensitive queries.[5] Therefore, we can conclude that apps are interested in knowing about the presence of other installed apps, and that, if Android blocked the two methods `getIA()` and `getIP()`, nosy apps would likely make use of other methods presented in Section IV.

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

[5]We conduced a similar analysis for paid apps, at a smaller scale, and found that paid apps are less likely to query for `LIAs`, probably because they rely less on third-party libs, especially ad libs. See Appendix A for details.

D. Analysis of Privacy Policies

Google's privacy-policy guidelines require apps that handle personal or sensitive user data to comprehensively disclose how the apps collect, use and share user-data. An example of a common violation, included in these guidelines, is "*An app that doesn't treat a user's inventory of installed apps as personal or sensitive user data*" [42]. In this section, we explain what developers understand about the guidelines and the declared purposes of the collection of `LIAs` by apps. For the sake of simplicity, in this section, we define a *nosy app* as an app that calls at least one sensitive method in the static (Section V-B) *or* dynamic (Section V-C) analysis .

As mentioned in Section V-A, out of 2917 apps in our dataset, we found and collected 2499 privacy policies. From 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 or paragraph to express the intention of collecting `LIAs`, *e.g.,* collect, gather, package, ID and software. 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, *e.g.,* "installed app", "app ID" and "installed software". Thereafter, we read the matched sentences and the corresponding privacy policy. During the process, we discovered new patterns and we updated the expression list. We stopped when the results converged.

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, *i.e.,* only 7.7% of the nosy apps inform the users about their collections of `LIAs`. Interestingly, some apps have exactly the same privacy policies, even though they are from different companies (*e.g.,* [47] and [48]). Also, 33 apps mentioned the collections of `LIAs`, but we did not find them in both static and dynamic analysis. For these apps, we performed a more thorough dynamic analysis, *i.e.,* we used them as a normal user, 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.

Among the generic declared purposes of the collections of `LIAs` by apps, *e.g.,* for improving the service (*e.g.,* [49]) or for troubleshooting the service in case the app crashes (*e.g.,* [50]), some apps explicitly declare that they collect `LIAs` for targeted ads (*e.g.,* [51], [52]), even more specifically targeted ads by third-party ad networks (*e.g.,* [53]). Unexpectedly, we have found that among the 162 privacy policies that mention the collections of `LIAs`, 76 of them categorize `LIAs` as *non-personal* information, whereas Google defines this as personal information. This shows a serious misunderstanding between developers and Google's guidelines.

VI. Existing Protection Mechanisms

There are no robust mechanisms available to help users hide the existence of sensitive apps from nosy apps. Below, we present some mechanisms that can offer partial protection.

Mechanisms by Google. Android does not provide users with a mechanism to hide the existence of apps from other apps. However, users could repurpose existing Android mechanisms for partially hiding apps. For instance, Android supports *Multiple Users* [54] on a single phone by separating user accounts and app data. Sensitive apps could then be installed and used in one or more secondary accounts. This approach will prevent nosy apps in the primary account from learning which apps are installed in secondary accounts, if nosy apps use the `getIA()` and `getIP()` methods. Unfortunately, nosy apps only need to include additional parameters or use different methods, that also do not require permissions, to bypass the isolation provided by multiple user accounts. We found several ways to identify which apps are installed in secondary accounts, see more details in the Appendix B. Multiple user accounts also have a negative impact on functionality and usability. For example, while the primary account is in the foreground, apps in secondary accounts cannot provide notifications to users or use Bluetooth services (important for mHealth apps). Moreover, apps in different accounts cannot be used simultaneously without switching accounts, an operation that introduces a noticeable delay.

Android for Work [55], a solution that separates work apps from personal apps, provides better usability but similar level of protection as multiple users, as it relies on the latter to provide app isolation. Our tests (Appendix B) 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* [56], that 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 external Bluetooth-enabled devices [57]. Our solution (Section VII) uses this idea of running an app without installing it to hide its presence.

Google classifies the `LIA` as *personal* information and, therefore, requires apps that collect `LIAs` to include in their privacy policies the purpose for collecting this data. Apps that do not follow this requirement can then be classified as Potentially Harmful Apps (PHAs) or Mobile Unwanted Softwares (MuWS) [13], [14]. Android security services, such as Google Play Protect [58] and SafetyNet [59], periodically scan users' phones and could warn users if apps are behaving as PHAs or MuWS. Such mechanisms, however, do not seem to effectively protect against the unauthorized collection of `LIAs`. Our analyses show that only 7.7% of the apps declare their collections of `LIAs` in their privacy policies and some claim that a `LIA` is non-personal information (Sections V). Moreover, these mechanisms may fail to detect more targeted attacks, *e.g.,* a nosy app checking if a small subset of sensitive apps exists in the phone.

Mechanisms by Third-Parties. Samsung Knox [60] relies on secure hardware to offer isolation between work and personal

data and apps, similar to Android for Work. Hence, it could be use to hide sensitive apps. Unfortunately, we were not able to evaluate how robust is the protection offered by Knox w.r.t. hiding apps, as 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 hide apps from other apps in a different isolated environment, but not from apps in the same environment, *e.g.,* apps in the same isolated environment may come from different, untrusted sources. That is, a solution that provides isolation-per-app is preferable.

There are apps in the Google Play store that help users to hide the icons of their sensitive apps from the Android app launcher, (*e.g.,* [61]). Even though they help to hide the presence of the app from other human users (*e.g.,* nosy partners), the sensitive apps are still visible to other apps on the phone, through public APIs. Along the lines of user-level virtualization techniques (see Section II), we have found apps on the Google Play store that use these techniques to enable users to run multiple instances of an app on their phones in parallel and partially hide apps, (*e.g.,* [62]–[64]). However, these apps require the app to be installed first in the phone before protecting it, thus triggering installation and uninstallation broadcast events that can be detected by a nosy app (see Section IV). Moreover, users have to grant all the Android permissions to these (untrusted) apps, thus providing these apps with complete control over the phone. In addition, these apps provide only a single isolated space, *i.e.,* they do not protect 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.

VII. HideMyApp: A Practical Solution

In this section, we present HideMyApp (HMA), a system that hides the presence of sensitive apps.

A. System Model

The scenario envisioned for HMA is as follows. A hospital or a hospital consortium (hereafter 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.

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, such as the Google Play store. To allow the HMA Manager to install apps downloaded from the HMA App Store, similarly to other Google Play store alternatives such as Amazon [65] and F-Droid [66], users need to enable the "allow apps from unknown sources" setting on their phones. Also, starting from Android 8.0, Google has made this option more fine-grained by turning it into the "Install unknown apps" permission [67]. That is, users will need to grant this permission to the HMA Manager app to allow it to install apps from the HMA App Store.

B. Adversarial Model

We assume the Android OS on the users' phones to be trusted and secure, including the Linux kernel and the Java API framework. We assume there is a nosy app that wants to learn if a specific app is present on the phone, and the nosy app can have all permissions and it can manage to have adb privilege, *e.g.,* by using methods presented by Lin *et al.* [41]. As a result, the nosy app can have access to all static and dynamic information related to installed apps on the phone, as presented in Section IV. We assume that the HMA App Store and the HMA Manager provided by the hospitals 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 IX-B.

C. Design Goals

The goal of HMA is to effectively hide the presence of sensitive apps, while preserving their usability and functionality.

(O1) Privacy protection. It should be difficult for a nosy app to fingerprint and identify sensitive apps on the same phone.

(O2) No firmware modifications. The solution should run on stock Android phones, *i.e.,* it should not require the phones to run customized versions of Android firmware, such as extensions to Android's middleware or kernel. This also means that the solution should not require to root the phones.

(O3) Preserving the app-isolation security model of Android. Each app should have its own private directory and run in its own dedicated process.

(O4) No app modifications. The solution should not require to access to apps' source-code, *i.e.,* only APK files are needed.

(O5) Usability. The solution should preserve the key functionality of sensitive apps and their usability.

D. 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 random package name and obfuscated app components, hence, nosy apps cannot fingerprint a sensitive app based on the information they can collect 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 (*e.g.,* [21]), *i.e.,* 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 VIII-A). Through the HMA Manager app, users can (un)install, open and update sensitive apps w/o being discovered by the OS and other apps. We detail these operations in Section VIII-B.

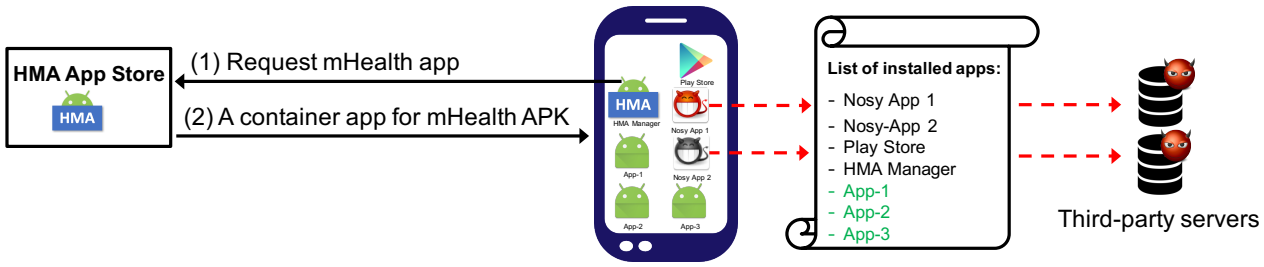


Fig. 1: Overview of the HMA architecture. Nosy apps would only learn the random names of the container apps.

VIII. HMA System Description

In this section, we detail the design of HMA, including its system-bootstrapping procedure and operations.

A. HMA System Bootstrapping

To bootstrap the system, the hospitals need to set up the HMA App Store and provide users with the HMA Manager app.

1) HMA Manager App: Recall, to hide their presence, sensitive apps are not registered in the OS; instead, their container apps are registered. Consequently, if users open their default Android app launcher, they will not see their sensitive apps; instead, they will see container apps with generic icons and random names. To solve this usability issue, at installation time (Section VIII-B1), the HMA Manager app keeps track of the one-to-one mappings between sensitive apps and their container apps. Based on 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, *e.g.,* using an anonymous proxy or Tor.

2) HMA App Store: The HMA App Store receives app-installation and app-update requests from HMA Manager apps, accordingly creates container apps for the requested apps and sends them to the HMA Manager apps. 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 VIII-B.

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

- It creates an empty app with a generic app icon, a random package name and label, and it imports in to 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 (*i.e.,* 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 IV). To mitigate this problem, for each activity and service declared by the sensitive app, the container app declares a randomly-named component. At runtime, the container app will map these random names to the real names. For broadcast receivers, the container app dynamically registers them at runtime. The case of content providers is discussed in Section IX.
- The HMA App Store compiles the container app to obtain its APK and signs it.

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

HMA User-Level Virtualization. To launch the APK of a sensitive app without installing it, its container app spawns 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 (*i.e.,* the OS and the app framework). To do so, we rely on the technique implemented by DroidPlugin [68], an open-source lib for app virtualization. We explain DroidPlugin techniques in Appendix C.

B. HMA Operations

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

1) 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 an installation request consisting of the name of the sensitive app to the HMA App Store. The HMA App Store creates a container app for the requested sensitive app (see Section VIII-A2) 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. In addition, the

HMA Manager saves a record of the package name of the container app and the package name, the original icon and label of the sensitive app in a database in its private directory.

2) 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 VIII-B1), 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, and stores the APK in its private directory. This incurs some delays to the first launch of sensitive apps, but it is needed, to prevent sensitive apps from being fingerprinted. This is because, if the sensitive APK was included in the resources or assets folders of its container app so that the container app can 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 IV). Also, Android does not allow apps to automatically start their background services upon their 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 (*i.e.,* the OS and the Android app framework), as described in Section VIII-A2. 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. As a result, they will be shown with the random package name of the container app. This, however, does not affect the usability and comprehensibility of the permission requests, as shown by our user study (Section X-E).

3) App Update: To check for updates, at predefined time intervals (*e.g.,* every day), the HMA Manager app anonymously sends to the HMA App Store the list of sensitive apps on the user's phone. Alternatively, the HMA App Store can send a push notification to all HMA Manager clients when there is an update for an app. If an app has an update, the HMA Manager sends the package name of its existing container app to the HMA App Store. Using this existing package name, the HMA App Store creates a new container app for the updated sensitive app. Note that 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 Manager receives the updated container app from the HMA App Store, and it prompts the user for her confirmation about the installation. Once the user accepts, the updated container app is installed on the phone, similarly to standard app-update procedure on Android (*i.e.,* the updated app reuses its existing directory and UID).

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

IX. Privacy and Security Analysis

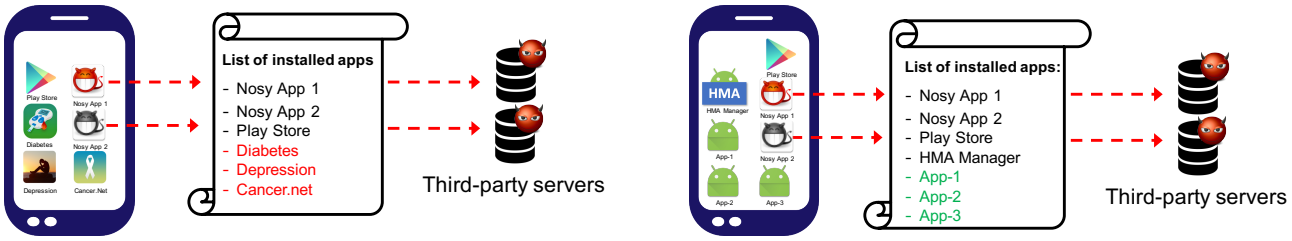
Here, we present an analysis of HMA to show that it effectively achieves its privacy and security goals (Section VII-C).

A. Privacy

Fig. 2 compares the information about sensitive apps that a nosy app can learn w/ and w/o HMA. We will explain how HMA achieves its privacy goals in detail below.

Static Information. Obviously, HMA effectively protects the *mandatory* information of sensitive apps by obfuscating the metadata associated with the sensitive app. 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 random package name is installed. With adb privilege, the nosy app can obtain the path to the APK of the container app, but this path does not contain any information about the sensitive app. Moreover, the nosy app can retrieve the APK file of the container app, but it cannot retrieve the APK file of the sensitive app, because the sensitive APK is not included in the container-app's APK (it is stored inside the private directory of the container app). Second, the resources, shared libraries, developers' signature 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 file, *i.e.,* they are dynamically loaded from the sensitive app's APKs at runtime. Third, the nosy app cannot learn the components' names of the sensitive app, because these names are randomized. To prevent a nosy app from being able to fingerprint sensitive apps based on the number of app components declared in their container apps, during the generation of container apps, the HMA App Store adds dummy random components such that all container apps generated by the HMA App Store declare the same number of random components.

Regarding *optional* information, a nosy app might try to fingerprint sensitive apps based on the set of permissions declared by their container-apps. This problem can be mitigated if all container apps declare a union of permissions requested by sensitive apps in the HMA App Store. Note that if the device's OS is at least 6.0, container apps only request, at runtime, for permissions needed by their sensitive apps, and users can grant or decline the requests, thus making it difficult for nosy apps to fingerprint sensitive apps based on the set of permissions that their container apps have been granted. In addition, to hide their presence, sensitive apps should not expose themselves to other apps by using content providers and implicit intents (Section XI-A). Moreover, to prevent a nosy app from fingerprinting sensitive apps based on their customized configurations, such as 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, and the same



(a) w/o HMA (b) w/ HMA

Fig. 2: Information learnt by nosy apps if sensitive apps are executed (a) w/o HMA and (b) w/ HMA.

approach has been used in other deployed systems, such as in Tor where all the browsers have the same default window size and user-agent strings [69]. 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.,* [70] and [71]).

Runtime Information. A nosy app cannot fingerprint a sensitive app based on the the list of running processes it can retrieve, because the sensitive app runs inside the child process of its container app with a random name. To prevent nosy apps from fingerprinting sensitive apps based on statistics about resources consumed by their container apps (including network and storage consumption, power consumption, and memory footprints, system logs and diagnostic outputs), the container apps can randomly generate dummy data to obfuscate the usage statistics of sensitive apps. The container apps can also obfuscate the UI states of the sensitive apps by overlaying transparent frames on the real screens of the sensitive apps. Regarding files in the phone's external storage, container apps can intercept and translate calls from sensitive apps when they create or access files in the external storage. However, note that apps are not recommended to store data in external storage, especially mHealth apps. In addition, to prevent the nosy app from receiving notifications sent to other apps, the users should not grant the BIND_NOTIFICATION_LISTENER_SERVICE to apps that they do not trust.

B. Security

As explained in Section VIII-A2, the user-level virtualization technique used by HMA to launch an APK in HMA does not require users to modify the OS of the phone. The Android's app-sandboxing 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 [13], [58]. Also, recently, this setting was converted to a per-app permission; hence, it does not propagate to other apps [67].

Furthermore, with HMA, developers still control and sign their apps, and the HMA App Store validates these signatures before publishing the apps on the app store. The container apps are signed by the HMA App Store, vouching for the developers' signatures. Furthermore, HMA container apps only ask users for permissions requested by sensitive apps, unlike solutions, *e.g.,* Boxify [21], that require all permissions to be granted beforehand. 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 specific user.

X. 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) overheads experienced by mHealth apps, (2) HMA runtime robustness and its compatibility with mHealth apps, and (3) HMA usability.

A. Dataset

We selected 50 apps from the medical category on the Google Play store, of which 42 apps are free and 8 apps are paid. To have a diverse dataset, we selected apps based on their popularity (*i.e.,* 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 can be supported, similarly to other services, the only challenge is engineering efforts. We also filtered out apps that use Facebook SDKs, because such SDKs often use custom layouts, which have not been yet supported by the user-level virtualization library that HMA uses (*i.e.,* DroidPlugin [68]). Exploring the interaction mechanisms between custom layouts and custom notifications with the Android app framework is an avenue for future work.

After filtering, we narrowed down the dataset to 30 apps (24 free apps and 6 paid apps, see Appendix D). Our dataset

is somewhat diverse with 15 medical conditions.Also, apps in our dataset support features that are crucial for mHealth apps, such as Bluetooth connections with external medical devices (*e.g.,* `Beurer HealthManager` app [72]) and Internet connections (*e.g.,* `Cancer.Net` app [73]).

B. Implementation Details

Our prototype implemented 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 it relied on an open-source library called DroidPlugin [68] for the user-level virtualization. Our prototype container apps dynamically loaded the classes and resources of the apps from the mHealth APKs, and they supported the interception and proxy of API calls that are commonly used and crucial to mHealth apps, such as APIs related to Bluetooth connections and SQLite databases.

C. Performance Overhead

In this section, we present the delays introduced by `HMA` to sensitive apps during app-installation and app-launch operations. We omit the app-update operation, because app-update and app-installation operations are similar. For the evaluation of delays added by the user-level virtualization to commonly used API methods and system calls at runtime of sensitive apps, we refer the readers to existing work such as Boxify [21]; they show that such overhead is negligible, *e.g.,* 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, one of the most recent Android versions. In our experiments, the `HMA App Store` connected to the phone through a micro-USB cable, thus network delays were not considered. However, note that, compared to the standard use of apps, `HMA` incurs negligible network-delay overheads, because the only payload overhead introduced by `HMA` is the container app, whose size is of several hundreds of kilobytes only.

1) App Installation: When a user wants to install an mHealth app, the `HMA App Store` first needs to create 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. The size of the container apps is only several hundreds of kilobytes, which would take less than a second for the `HMA Manager` app to download (assuming a 3G or 4G Internet connection). As a result, the total delay incurred by `HMA` will take less than 5 s, in the *worst-case* scenario; This an acceptable one-time delay.

2) App Launch: On Android, apps can be launched from different states, *i.e., 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. To measure cold-start delays (*i.e.,* the time elapsed since a user clicks on the app's icon until the first screen of the app is loaded), we rely on the Android's official launch-performance profiling method [74]. For each app, we installed its container app, copied its APK file to the container-app's private directory and launched the container app through `adb`. Thereafter, we 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 delete all data associated with the app [35] (*i.e.,* 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 baseline measurements, we measured the delays experienced by mHealth apps when they were executed w/o `HMA`, *i.e.,* we installed the apps on the phone and followed the same procedure.

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 [75]. To simulate a warm start, we used the command `input keyevent 187` to bring the app to the background, and then 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. It is intuitive that, in `HMA`, the first launch of an mHealth app will experience more delays than the subsequent cold starts, because the container app has to download the APK from the `HMA App Store` and stores the APK in its private directory. It also needs to process the APK and cache needed information for the user-level virtualization. Our experiments show that the median of the first launch delays is 6.5 s (as compared to 0.8 s if the mHealth apps were launched w/o `HMA`). However, this delay occurs only once, and therefore it is negligible w.r.t. the lifetime of the app on the phone.

Fig. 3 shows the bar plot of subsequent cold-start delays experienced by mHealth apps when they are executed w/ and w/o `HMA`; 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 s and 1.3 s if the apps are executed w/ `HMA` and w/o `HMA` respectively. For 90% of the cases, the average delay with `HMA` is less than 2 s.

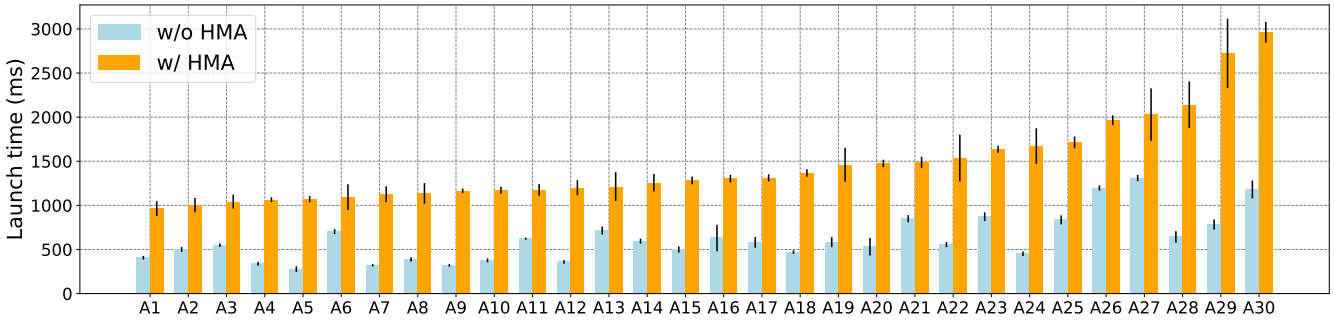


Fig. 3: Cold-start delays experienced by mHealth apps when they are executed w/ and w/o HMA. Note that our HMA implementation is a proof-of-concept, hence not-optimized. The heights of the bars represent mean values and the error bars represent the standard deviation. For each setting, we collected 50 measurements per app. The full names of the apps can be found in Appendix D.

Note that our prototype implementation is a proof-of-concept hence not optimized; a more optimized implementation of the system is left for future work. However, the observed delays are still under the delay limit suggested by Android (*i.e.,* 5 s [74]). Also, based on our user study (Section X-E), 97% of participants agreed that a launch delay of 5 s (*i.e.,* the delay experienced by the `Cancer.Net` app [73] in its first launch w/ HMA) is acceptable.

Regarding the warm-start delays, we found that the average warm-start delays experienced by our tested app when it was launched w/ and w/o HMA were both approximately 550 ms. 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 unfortunately cannot simulate this case, because Android does not provide methods to control the garbage collector. However, in that case, the delay will still be less than cold-start delays (*i.e.,* at most 3 s, see above).

D. 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.,* [23] and [76]), we manually tested each app in our data-set with HMA. For each mHealth app, we extracted its APK and used the `HMA App Store` to create a container app 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 crashed during its execution. We found that all of the apps in our dataset work normally, except one app that threw an error when making an SQLite connection. However, we ran an example app from [77] that uses the official Android APIs for database (*i.e.,* `android.database.sqlite`) to insert and retrieve records from an SQLite database and the app ran successfully. We suspected the app specified the full path to the directory of the database (*i.e.,* `/data/data/package-name/db-name`), hence failing the call, because the directory does not exist. This problem could be solved if the developers specified 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 and found that HMA can be successfully deployed on mainstream commercial Android devices. However, 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 have not supported were called in the later versions of Android. This problem, however, can be solved if these services are hooked, as we discussed in Section X-B.

E. HMA Usability and Desirability

To evaluate the usability of HMA and the users' interest for it (*i.e.,* desirability), we conducted a user study at the laboratory for behavioral experiments at our institution.[6] Our study involved 30 student subjects (19 males, 11 females) with 18 different majors. The participants were experienced Android users, *i.e.,* 87% of the participants have been using an Android phone for at least one year, and they were relatively concerned about their privacy, *i.e.,* using the standard metric for measuring users' privacy perception (IUIPC) [78], we found that, on a scale from 1 to 5 for privacy postures, 97% of participants graded at least 3.0 and an average of 4.1.

We began the study with an entry survey (see transcript) [79] about demographics information, privacy postures and 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 (one app is a popular app about public-transportation timetables in the city where the university is located and one app is an mHealth app) with and without HMA, in order to precisely measure the users' perception of the delay introduced by HMA. The participants were provided with detailed instructions [79], including screenshots, on how to install and run the apps with and without HMA.

[6]Our user study was approved by our institutional ethical committee.

67% of the participants used the first app in the past, whereas only 7% of them used the second one (or an app of the same category). We finished the user study with an exit survey containing questions related to the usability of the solution and the users' levels of interest in HMA. The user-study session took approximately 45 minutes and we paid each participant ∼$25. The full transcript of survey questions and the user-study instructions can be found at [79]

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 to be collected by mobile 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 have used health-related apps. Indeed, our study confirms the findings from previous works (*e.g.,* [16]) that the majority of people never read privacy policies; as a consequence, the current solution using privacy policies by Google for LIA collections is not satisfactory. These findings show the need for a solution to hide the presence of a sensitive mobile app.

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 s and 5.1 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 with a 5-point Likert scales, we made the following observations. Almost all participants agreed that these changes and delays are acceptable (97% and 93% of the participants, respectively). In addition, 93% of the participants agreed that the use of 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 it protects and that they are at least somewhat interested in using HMA. These results suggests that HMA is usable and desirable.

As most lab experiments, our study has a low ecological viability and some limitations which we acknowledge here. It should be noted, however, that the goal of our study was not to evaluate the design and the usability of the solution as a final product but rather to evaluate, based on a proof of concept with a basic UI, the interest and the perception of users regarding the general approach and its technical implications (*e.g.,* delays). First, the participants used phones provided by the experimenters, with only two apps and in a hypothetical scenario. Second, the users were provided with detailed instructions; therefore, we did not test how intuitive the proposed solution is. Finally, when they took the exit survey, the participants had already been briefed about the privacy problems of LIA, which might introduce a bias. We intend to conduct long-term studies and to refine and evaluate the design of the user interface and the global user experience.

XI. Discussions

A. HMA Limitations

As explained in Sections IX and X, HMA effectively hides the mandatory features and runtime information of sensitive apps. However, due to their hiding goal, optional features that help sensitive apps to intentionally expose themselves to other apps, including content providers and implicit intents, should not be used. This can limit their data-sharing capabilities; A possible workaround for this problem is to put apps that want to share data with each other in the same container app. In addition, due to the limitation of user-level virtualization library that HMA uses, if sensitive apps use many customizations with third-party SDKs, the container apps might fail to launch them. This might be a problem with complex apps, such as dating apps, but mHealth apps are unlikely to require many customizations. In the scenario envisioned by HMA, developers build apps for the hospitals, thus the HMA App Store can provide developers with a guideline about supported features so that their apps are HMA-compatible. To facilitate guideline compliance, the HMA App Store can also provide developers with IDE plugins to help them write compliant code; such approach has been proposed in existing work (*e.g.,* [70], [71]). Lastly, if the number of apps in the HMA App Store is small, a nosy app could guess the name of the sensitive app. To increase the anonymity-set, the hospitals can include simpler, non-sensitive apps (*e.g.,* informational) in the HMA App Store, and the popularity of apps on the store should not be public.

B. An Alternative Scenario

HMA is designed to work for the case of hospitals hosting their mHealth apps. However, there are other categories of apps that can be considered sensitive (*e.g.,* apps about religion or sexual orientation). For example, in some countries, people would be arrested if they use a gay dating app [80]. Moreover, developers might publish their mHealth and other sensitive apps on app stores other than the HMA App Store, such as the Amazon or Google Play app stores. In such cases, the community or an organization can run a proxy service that retrieves the APKs of these apps from these stores and create container apps for them accordingly; this service can be free or paid. This might require an agreement between the proxy service and the app developers and also, as discussed in Section XI-A, some apps might have optional features that are not recommended or features that are not supported by user-level virtualization techniques. Similarly to the case of the Amazon app store, the proxy service can provide developers with a testing service to test the compatibility of their apps.

XII. Conclusion

In this paper, we have systematically investigated the problem of apps fingerprinting other installed apps. We showed that apps can collect a significant amount of static and runtime information about other apps to fingerprint them. We also quantified the prevalence of the problem and have shown that third-party libs and apps are interested in learning the list of

installed apps on the phone. Moreover, our analysis showed that there are no existing mechanisms to hide the presence of a sensitive app from other apps. Therefore, we have proposed HMA, the first solution that addresses this problem. HMA does not require any modifications to the mobile OS and it preserves the key functionality of apps. Our thorough evaluation of HMA on a diverse data-set of both free and paid mHealth apps from the Google Play store shows that HMA is practical with reasonable operational delays. Moreover, the results of our user study suggest that HMA is perceived as usable and of interest to the users. For future work, we plan to implement a full prototype of the system and make it publicly available.

REFERENCES

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

[2] "The mHealth apps market is getting crowded." https://research2guidance.com/mhealth-app-market-getting-crowded-259000-mhealth-apps-now/, last visited: Jan. 2018.

[3] D. Kotz, C. A. Gunter, S. Kumar, and J. P. Weiner, "Privacy and security in mobile health: A research agenda," *Computer*, vol. 49, no. 6, June 2016.

[4] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Predicting user traits from a snapshot of apps installed on a smartphone," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 18, no. 2, Jun. 2014.

[5] S. Seneviratne, A. Seneviratne, P. Mohapatra, and A. Mahanti, "Your installed apps reveal your gender and more!" *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 18, no. 3, 2015.

[6] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for all! assessing user data exposure to advertising libraries on android," in *Proc. of the 2016 Network and Distributed System Security Symposium*, 2016.

[7] J. P. Achara, G. Acs, and C. Castelluccia, "On the unicity of smartphone applications," in *Proc. of the 2015 ACM Workshop on Privacy in the Electronic Society*, 2015.

[8] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from android public resources," in *Proc. of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[9] "Twitter scanning users' other apps to help deliver 'tailored content' ," https://www.theguardian.com/technology/2014/nov/27/twitter-scanning-other-apps-tailored-content, last visited: Jan. 2018.

[10] "About Twitter's app graph," https://help.twitter.com/en/safety-and-security/app-graph.

[11] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. of the 2012 ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[12] "Report: Smartphone owners are using 9 apps per day, 30 per month," https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/, last visited: Feb. 2018.

[13] "Android Security 2016 Year In Review," https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf, last visited: Feb. 2018.

[14] "Android Security 2015 Year In Review," https://source.android.com/security/reports/Google_Android_Security_2015_Report_Final.pdf, last visited: Feb. 2018.

[15] "Android Developer Policy Center," https://play.google.com/about/developer-content-policy-print/, last visited: Feb. 2018.

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

[17] "Additional protections by Safe Browsing for Android users ," https://security.googleblog.com/2017/12/additional-protections-by-safe-browsing.html, last visited: Jan. 2018.

[18] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: An empirical study of ssl warning effectiveness," in *Proc. of the 2009 USENIX Security Symposium*, 2009.

[19] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proc. of the 2012 Symposium on Usable Privacy and Security*, 2012.

[20] R. Xu, H. Saidi, and R. J. Anderson, "Aurasium: practical policy enforcement for android applications." in *Proc. of the 2012 USENIX Security Symposium*, 2012.

[21] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged app sandboxing for stock android." in *Proc. of the 2015 USENIX Security Symposium*, 2015.

[22] "HMA website," https://[obfuscatedforsubmission], last visited: Jan. 2018.

[23] S. Jaebaek, K. Daehyeok, C. Donghyun, S. Insik, and K. Taesoo, "FLEXDROID: enforcing in-app privilege separation in android," in *Proc. of the 2016 Network and Distributed System Security Symposium*, 2016.

[24] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proc. of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, 2014.

[25] M. Naveed, X.-y. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside job: Understanding and mitigating the threat of external device mis-binding on android." in *Proc. of the 2014 Network and Distributed System Security Symposium*, 2014.

[26] E. Malmi and I. Weber, "You are what apps you use: Demographic prediction based on user's apps," in *Proc. of the 2016 International AAAI Conference on Web and Social Media*, 2016.

[27] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, "Networkprofiler: Towards automatic fingerprinting of android apps," in *Proceedings IEEE INFOCOM*, 2013.

[28] Q. Xu, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, A. Nucci, and T. Andrews, "Automatic generation of mobile app signatures from traffic observations," in *IEEE Conference on Computer Communications (INFOCOM)*, 2015.

[29] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016.

[30] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Robust smartphone app identification via encrypted network traffic analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 1, pp. 63–78, Jan 2018.

[31] G. G. Gulys, G. Acs, and C. Castelluccia", ""near-optimal fingerprinting with constraints"," *"Proceedings on Privacy Enhancing Technologies"*, vol. "2016", no. "4", pp. 470 – 487, "2016".

[32] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna, "Njas: Sandboxing unmodified applications in non-rooted devices running stock android," in *Proc. of the ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015.

[33] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The art of app compartmentalization: Compiler-based library privilege separation on stock android," in *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[34] "Permissions Overview," https://developer.android.com/guide/topics/permissions/overview, last visited: May 2018.

[35] "Android Debug Bridge (adb)," https://developer.android.com/studio/command-line/adb.html, last visited: Dec. 2017.

[36] "Application Fundamentals," https://developer.android.com/guide/components/fundamentals, last visited: May 2018.

[37] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic, "Appscanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 439–454.

[38] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into your app without actually seeing it: Ui state inference and novel android attacks." in *USENIX Security Symposium*, 2014, pp. 1037–1052.

[39] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 143–157.

[40] Y. Chen, X. Jin, J. Sun, R. Zhang, and Y. Zhang, "Powerful: Mobile app fingerprinting via power analysis," in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 2017, pp. 1–9.

[41] C.-C. Lin, H. Li, X.-y. Zhou, and X. Wang, "Screenmilker: How to milk your android screen for secrets." in *NDSS*, 2014.

[42] "Privacy, Security, and Deception," https://play.google.com/about/privacy-security-deception/personal-sensitive/, last visited: Dec. 2017.

[43] "A tool for reverse engineering Android apk files," https://ibotpeaches.github.io/Apktool/, last visited: Dec. 2017.
[44] http://www.zdnet.com/article/accuweather-caught-sending-geo-location-data-even-when-denied-access/, 2017, last visited: Sep. 2017.
[45] "Trail: The Reflection API," https://docs.oracle.com/javase/tutorial/reflect/, last visited: Dec. 2017.
[46] "XPrivacy," https://github.com/M66B/XPrivacy, last visited: Dec. 2017.
[47] "Solitaire: Super Challenges," https://play.google.com/store/apps/details?id=com.cardgame.solitaire.full, last visited: Jan. 2018.
[48] "DH Texas Poker - Texas Hold'em," https://play.google.com/store/apps/details?id=com.droidhen.game.poker, last visited: Jan. 2018.
[49] "MX Player," https://play.google.com/store/apps/details?id=com.mxtech.videoplayer.ad, last visited: Dec. 2017.
[50] "Sweet Selfie - selfie camera, beauty cam, photo edit," https://play.google.com/store/apps/details?id=com.cam001.selfie, last visited: Dec. 2017.
[51] "InstaSize Editor: Photo Filters and Collage Maker," https://play.google.com/store/apps/details?id=com.jsdev.instasize, last visited: Dec. 2017.
[52] "Angry Birds," https://play.google.com/store/apps/details?id=com.rovio.angrybirds, last visited: Dec. 2017.
[53] "Neon Motocross," https://play.google.com/store/apps/details?id=com.motomex.neonmotocross, last visited: Dec. 2017.
[54] "Supporting Multiple Users — Android Open Source Project," https://source.android.com/devices/tech/admin/multi-user, last visited: May 2018.
[55] "Put Android to work," https://www.android.com/enterprise/employees/, last visited: Dec. 2017.
[56] "Android Instant Apps," https://developer.android.com/topic/instant-apps/index.html, last visited: Dec. 2017.
[57] "Android Instant Apps API reference," https://developer.android.com/topic/instant-apps/reference.html#instantapps.InstantApps, last visited: Dec. 2017.
[58] "Help protect against harmful apps with Google Play Protect," https://support.google.com/accounts/answer/2812853?hl=en, last visited: Jan. 2018.
[59] "Protecting against Security Threats with SafetyNet," https://developer.android.com/training/safetynet/index.html, last visited: Jan. 2018.
[60] "Samsung Knox," https://www.samsungknox.com/en, last visited: Dec. 2017.
[61] "Nova Launcher," https://play.google.com/store/apps/details?id=com.teslacoilsw.launcher&hl=en, last visited: Jan. 2018.
[62] "Parallel Space - Multiple accounts and Two face," https://play.google.com/store/apps/details?id=com.lbe.parallel.intl&hl=en, last visited: Jan. 2018.
[63] "Hide App, Private Dating, Safe Chat - PrivacyHider," https://play.google.com/store/apps/details?id=com.trigtech.privateme&hl=en, last visited: Jan. 2018.
[64] "Private Zone - Safe Vault," https://play.google.com/store/apps/details?id=com.leo.appmaster, last visited: May 2018.
[65] "Amazon App Store," https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011, last visited: Jan. 2018.
[66] "F-Droid," https://f-droid.org/en/, last visited: Jan. 2018.
[67] "Publish Your App," https://developer.android.com/studio/publish/index.html#publishing-unknown, last visited: Jan. 2018.
[68] "DroidPlugin," https://github.com/DroidPluginTeam/DroidPlugin, last visited: Jan. 2018.
[69] "The Design and Implementation of the Tor Browser," https://www.torproject.org/projects/torbrowser/design/, last visited: Jan. 2018.
[70] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting android developers in writingsecure code," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1065–1077.
[71] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "Flowfence: Practical data protection for emerging iot application frameworks." in *USENIX Security Symposium*, 2016, pp. 531–548.
[72] "Beurer HealthManager," https://play.google.com/store/apps/details?id=com.beurer.connect.healthmanager, last visited: Jan. 2018.
[73] "Cancer.Net Mobile," https://play.google.com/store/apps/details?id=com.fueled.cancernet, last visited: Jan. 2018.
[74] "Launch-Time Performance," https://developer.android.com/topic/performance/launch-time.html, last visited: Jan. 2018.
[75] https://github.com/commonsguy/cw-omnibus/tree/master/Activities/Lifecycle, last visited: May 2018.
[76] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid: Dynamically enforcing enterprise policy on android devices." in *Proc. of the 2015 Network and Distributed System Security Symposium*, 2015.
[77] "SQLite via SQLiteOpenHelper," https://github.com/commonsguy/cw-omnibus/tree/master/Database/ConstantsROWID, last visited: Jan. 2018.
[78] N. K. Malhotra, S. S. Kim, and J. Agarwal, "Internet users' information privacy concerns (iuipc): The construct, the scale, and a causal model," *Information systems research*, vol. 15, no. 4, pp. 336–355, 2004.
[79] https://www.dropbox.com/sh/lo273jtx6jkbf1c/AAB1BtkBmBuNVOV13OAwDu-ha?dl=1, last visited: May 2018.
[80] https://www.washingtonpost.com/news/worldviews/wp/2014/09/12/could-using-gay-dating-app-grindr-get-you-arrested-in-egypt/?noredirect=on&utm_term=.470e8bc8f41c, last visited: May 2018.
[81] "DexClassLoader," https://developer.android.com/reference/dalvik/system/DexClassLoader.html, last visited: Jan. 2018.

APPENDIX A
ANALYSIS OF PAID APPS

To estimate if there are differences between free and paid apps w.r.t. collecting `LIAs`, we performed similar analyses (*i.e.,* static and dynamic) with a set of 28 popular paid-apps in the Google Play store. We chose popular paid-apps (*i.e.,* top paid apps) from different categories; the list of paid apps evaluated can be found on Table IV in Appendix E. We found that 17.8% of the paid apps include 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*). While 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 due to the fact that they rely less on third-party libs, particularly ad libs.

APPENDIX B
ANDROID MULTIPLE USERS AND ANDROID FOR WORK EVALUATION

We evaluated how robust Android's multiple users [54] support and *Android for Work* [55] when used to hide the presence of sensitive apps from other apps in the same device. For this evaluation, we used a Nexus 5 device with Android 6.0 and two Android emulator instances with Android 8.1 and Android 9.0, respectively. In these devices, we installed a nosy app in the primary account (personal profile for Android for Work) and several sensitive apps in the secondary partition (work profile for Android for Work). The main goal was to see if the nosy app could identify the sensitive apps in the secondary accounts/work profile.

Our results show that it is easy for a nosy app to bypass the hiding protection offered by multiple user accounts and Android for work. A nosy app can use the following techniques to bypass such protection:

- In Android 7 or earlier, including an additional parameter flag (MATCH_UNINSTALLED_PACKAGES) in `getIA()` and `getIP()` will reveal the apps installed in secondary user accounts. For Android for Work, this approach works even in Android 8.1.
- In Android 9 (latest version of Android), a nosy app can use API methods such as getPackageUid(), getPackageGidS(), getInstallerPackageName() or getApplicationEnabledSetting() as oracles to check if an app is installed in the

device, even if it is installed in a secondary account or in a work profile. The nosy app only need to include the package name of the targeted sensitive app as a parameter to these methods. If the targeted app is installed in the device, these methods will return valid values, otherwise they return an error message or null.

- A nosy app can guess the UIDs of the installed apps in all the accounts and work profiles, by looking at the /proc/uid directory to learn the ranges of current UIDs in the system, and then guessing UIDs of other apps in the device using the getNameForUid() method. This method will return a package name given a UID as a input parameter; if the app does not exist, it returns null. Hence, it can be used as an oracle to retrieve the `LIA` 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 in the device, independently of the account or profile it was installed, by using the shell command: *pidof* <*PackageName*>. This approach was tested on Android 9.
- A nosy app with ADB privilege can get the `LIA`, which includes apps in secondary accounts and work profiles, by using the shell command *dumpsys*. This approach was tested on Android 9.

APPENDIX C
DROIDPLUGIN USER-LEVEL VIRTUALIZATION

To enable the `ClassLoader` of the container app to load the classes and resources from the APK, the container app creates a `LoadedApk` object from the APK file (through the public API method `getPackageInfoNoCheck()`). A `LoadedApk` object is an in-memory representation of the APK file through which source-code and resources of the app can be obtained. Thereafter, the container app creates a customized `ClassLoader` object with information from the APK by using the constructor provided by the Android APIs [81], and it uses Java refection to set this newly created `ClassLoader` to the `mClassLoader` field in the aforementioned `LoadedApk` object. This `LoadedApk` object is added to the `mPackages` cache in the current `ActivityThread` object of the container app.

Once the APK is loaded, the container app needs to intercept and proxy API calls between the process of the sensitive app and system services and the system calls between the app and the OS. For the API calls, the container app has to hook the `Binder` communication between the sensitive app and system services, such as `ActivityManagerService`, `ClipboardManagerService` and `PackageManagerService`. In the Android framework, the `Binder` proxy object of a system service is accessed through the method `getService()` of the `ServiceManager`. For efficiency, when this function is called, the system first looks for the `Binder` proxy object in a `Map` object called `sCache`. Therefore, to hook a system service, the container app first creates a fake Binder proxy object and inserts this fake object in the `sCache Map` object of the system. This way, when the process of the sensitive app wants to access a system service, it will use the fake proxy object created by `HMA`, instead of the original one. Consequently, the container app can intercept the `Binder` communication between the sensitive app and the system. For system-call interceptions, we refer the readers to previous work *e.g.,* [21]; we omit the descriptions due to space constraints.

APPENDIX D
APPS TESTED WITH HMA

Table III.

APPENDIX E
EVALUATED PAID-APPS

Table IV.

Index	App Name	Package Name	# Downloads
1	My Ovulation Calculator	com.ecare.ovulationcalculator	1M - 5M
2	Blood Pressure Log - MyDiary	com.zlamanit.blood.pressure	500K - 1M
3	DreamMapper	com.philips.sleepmapper.root	100K - 500K
4	Breathing Zone	com.breathing.zone	5K - 10K
5	Alzheimer's Speed of Processing Game	com.TinyHappySteps.Bird	1K - 5K
6	Cancer.Net Mobile	com.fueled.cancernet	10K - 50K
7	AIDS Info Drug Database	com.aidsinfo.aidsinfoapp	5K - 10K
8	My Pain Diary	com.damonlynn.mypaindiary	5K - 10K
9	iBP Blood Pressure	com.leadingedgeapps.ibp	10K - 50K
10	Squeezy: NHS Pelvic Floor App	com.propagator.squeezy	10K - 50K
11	OneTouch Reveal	com.lifescan.reveal	500K - 1M
12	ADHD Adults	com.labshealth.tdahadults	10K - 50K
13	Baritastic - Bariatric Tracker	com.baritastic.view	100K - 500K
14	Mole Mapper	edu.ohsu.molemapper	1K - 5K
15	Respiroguide Pro	com.tremend.respiroguide	10K - 50K
16	FearTools - Anxiety Aid	com.feartools.feartools	10K - 50K
17	Back Pain Relieving Exercises	com.backpainrelieving.backpain	10K - 50K
18	OnTrack Diabetes	com.gexperts.ontrack	500K - 1M
19	Asthmatic	be.sarahvn.asthmatic	50 - 100
20	MoodTools - Depression Aid	com.moodtools.moodtools	100K - 500K
21	Pain Diary & Forum CatchMyPain	com.sanovation.catchmypain.phone	50K - 100K
22	Beurer HealthManager	com.beurer.connect.healthmanager	100K - 500K
23	Constant Therapy	com.constanttherapy.android.main	10K - 50K
24	Self-help Anxiety Management	com.uwe.myoxygen	100K - 500K
25	Pregnancy Calendar and Tracker	ru.mobiledimension.kbr	1M - 5M
26	BELONG Beating Cancer Together	com.belongtail.belong	10K - 50K
27	AsthmaMD	com.mobilebreeze.AsthmaMD	10K - 50K
28	Propeller	com.asthmapolis.mobile	5K - 10K
29	mySugr: the blood sugar tracker made just for you	com.mysugr.android.companion	500K - 1M
30	QuitNow! - Quit smoking	com.EAGINsoftware.dejaloYa	1M - 5M

TABLE III: mHealth apps tested with HMA.

1. org.xtramath.mathfacts	15. com.maxmpz.audioplayer.unlock
2. co.wordswag.wordswag	16. com.microphone.earspy.pro
3. com.vicman.photolabpro	17. com.tdr3.hs.android
4. com.ultimateguitar.tabs	18. com.etermax.preguntados.pro
5. com.burleighlabs.babypics	19. com.real.bodywork.muscle.trigger.points
6. com.ellisapps.itrackbitesplus	20. com.devolver.spaceplan
7. com.intsig.lic.camscanner	21. org.twisevictory.apps
8. com.period.tracker.deluxe	22. com.samruston.weather
9. com.mojang.minecraftpe	23. au.com.shiftyjelly.pocketcasts
10. com.wolfram.android.alpha	24. com.azumio.instantheartrate.full
11. slide.cameraZoom	25. udk.android.reader
12. com.digipom.easyvoicerecorder.pro	26. radiotime.player
13. com.melodis.midomiMusicIdentifier	27. com.flyersoft.moonreaderp
14. com.laurencedawson.reddit_sync.pro	28. kr.aboy.tools

TABLE IV: List of paid apps evaluated in our study. Only a few paid apps (17.8% static analysis, 7.4% dynamic analysis) seems to request information about LIAs (Section V).