IADIS International Journal on WWW/Internet
Vol. 22, No. 2, pp. 93-110
ISSN: 1645-7641

A STUDY OF PERSONAL INFORMATION LEAKS IN MOBILE MEDICAL, HEALTH, AND FITNESS APPS

Alireza Ardalani, Joseph Antonucci and Iulian Neamtiu
New Jersey Institute of Technology, Newark, USA

ABSTRACT

There has been a proliferation of mobile apps in the Medical, as well as Health&Fitness categories. These apps have a wide audience, from medical providers, to patients, to end users who want to track their fitness goals. The low barrier to entry on mobile app stores raises questions about the diligence and competence of the developers who publish these apps, especially regarding the practices they use for user data collection, processing, and storage. To help understand the nature of data that is collected, and how it is processed, as well as where it is sent, we developed a tool named PIT (Personal Information Tracker) and made it available as open source. We used PIT to perform a multi-faceted study on 2832 Android apps: 2211 Medical apps and 621 Health&Fitness apps. We first define Personal Information (PI) as 17 different groups of sensitive information, e.g., user's identity, address and financial information, medical history or anthropometric data. PIT first extracts the elements in the app's User Interface (UI) where this information is collected. The collected information could be processed by the app's own code or third-party code; our approach disambiguates between the two. Next, PIT tracks, via static analysis, where the information is "leaked", i.e., it escapes the scope of the app, either locally on the phone or remotely via the network. Then, we conduct a link analysis that examines the URLs an app connects with, to understand the origin and destination of data that apps collect and process. We found that most apps leak 1–5 PI items (email, credit card, phone number, address, name, being the most frequent). Leak destinations include the network (25%), local databases (37%), logs (23%), and files or I/O (15%). While Medical apps have more leaks overall, as they collect data on medical history, surprisingly, Health&Fitness apps also collect, and leak, medical data. We also found that leaks that are due to third-party code (e.g., code for ads, analytics, or user engagement) are much more numerous (2x–12x) than leaks due to app's own code. Finally, our link analysis shows that most apps access 20–80 URLs (typically third-party URLs and Cloud APIs) though some apps could access more than 1,000 URLs.

KEYWORDS

Personal Information Leaks, Medical Apps, Health & Fitness Apps, GUI Analysis, Android, Information Flow Analysis

1. INTRODUCTION

Mobile apps collect billions of users' data each day: Android alone has in excess of 3 billion monthly active users (Samat, 2022). Among these, apps designed for medical, health, or fitness purposes, are particularly important, because the data they collect contains personally identifiable information, and medical/health data. However, one major downside of apps collecting (and users providing) this data lies in the possibility of its misuse. Data is routinely sent to advertisers; stored in the Cloud where it can be accessed by malicious actors; alternatively, naive developers can store or save personal information in a way that enables other malicious apps to read, copy or leak otherwise confidential information. PI leaks are consequential: leaked PI can be sold for profit, (mis)used to identify individuals, or used as a non-repudiation authenticator for blackmail purposes (Van Alstin, 2024). Even in relatively mild cases, collecting just a person's date of birth, gender, and postal code can be enough to identify them (Sweeney, 2000).

Little effort has been put into understanding and exposing leaks of data that users supply via the app's UI, especially in the context of health, fitness, or medical apps. We devised a two-prong approach towards understanding and addressing this issue and implemented it in a tool named PIT. We construct an automatic way of extracting PI from app GUIs via Information Retrieval, and then couple this with a flow analysis to understand where, and how, PI is leaked. We ran PIT on 2,832 Android apps (overall: 47,749 GUI elements) and found 44,753 leaks. We categorized the nature of the leaks: 18.3% were due to app's own code, aka own-code, while the vast majority, 81.7%, were due to external-code (ads, analytics, user engagement, etc.). Medical apps have 3x more PI leaks, and 2.5x more network leaks, than Health&Fitness apps. Our own prior work (Ardalani et al., 2024) used a less precise mechanism for labeling UI fields as collection points for PI data, which could result in a higher rate of false positives compared to this work; in addition, our prior work did not look at per-app statistical analysis of how many PIs are leaked, and which PIs are leaked together. Finally, our prior study did not analyze embedded links (URLs). Prior work on PI leaks have used dynamic analysis (McClurg et al., 2013) but only analyzed up to 100 apps. Some efforts used differential analysis, a cryptographical method of retrieving a plaintext from an encrypted data stream (Continella et al., 2017). Other efforts used network traffic analysis (Jia et al, 2019) (Ren et al, 2016) but only examined data that left the device, without tracking sensitive data that could be exposed in device storage. Our work differs from prior works in what information we consider to be leaks, and how we characterize the leaks. One line of prior work aimed to find leaks of hardware device identifiers, such as MAC addresses, serial numbers, device location, etc. (Arzt et al., 2014). However, we aim to find leaks of personal information, or personal identifiers such as name, weight, height, date of birth, or postal (Zip) code. Another line of work on detecting sensitive input in UI (Huang et al., 2015) has included identity and weight/height information as we do, but did not look for medical information, and did not track where the information is flowing, or who is responsible for the leak (own-code vs external-code). Work that has distinguished between own-code (first-party) and external-code (third-party) leaks (Rahaman et al., 2021) has only focused on hardware identifiers, and not analyzed the destination of leaks, e.g., network, logs, files, local DB storage, as we do.

2. APPROACH

PIT consists of two analyses. First, PIT runs a GUI analysis that extracts and categorizes the PI collected. Second, PIT runs a taint (information flow) analysis to determine where the PI flows (leaks).

2.1 GUI Analysis

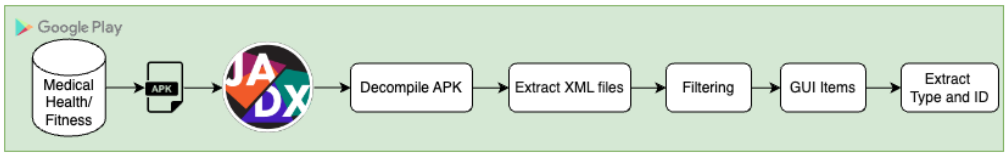

Figure 1. Overview of PIT's GUI analysis process

Overview. Figure 1 shows our analysis process. Android apps are distributed as ".apk" files, so we name them APKs for short. An APK bundles app code with app resources (e.g., strings, icons, images). The dataset for this study consists of 2,832 APK files, downloaded from the Google Play Store's Medical, as well as Health&Fitness, categories. The APK files are then decompiled using JADX (JADX, 2024), yielding the source code and associated resources, including XML files.

Extracting GUI elements. Next, the extracted XML files, which contain the GUI layout definitions and other critical information about the application's GUI components, are processed. In Android, all GUI elements descend from class *View*. We filter out non-*View* elements, ensuring that only relevant GUI components, such as *EditText*, *Spinner*, *CheckBox* and *RadioButton*, are retained.

Figure 2. UI-XML mapping for app *Weight Loss Tracker*

In Figure 2 we show the mapping between XML code and UI for app *Weight Loss Tracker*. This app contains several *Views*: *EditText*, *Button*, etc. Note how in this case the *android:id* associated with UI elements facilitates analysis, because the *id* name, e.g., *id/weightEditText* or *id/user_birthdate_button*, directly encodes the PI semantics. However, there were two main additional challenges associated with UI extraction. First, there are UI elements where the *android:id* is non-suggestive, which required us to analyze other XML attributes, e.g., text, or hints associated with that *View*. Second, especially for medical information, we had to look for adjacent terms, as explained under "PI-based categorization" below.

Personal Information (PI) definition. Regulatory frameworks such as HIPAA in the US define "protected health information" to include health conditions, care provided, and information that can be used to identify the individual (e.g., name, phone number, social security number, birthdate, etc. (HIPAA, 2024)). The focus of this paper is on information that is collected from the user, via the GUI. Specifically, based on initial analysis of the most frequent information present in GUIs, we focus on 17 PI grouped into:

- *Identity*: email, name (first, last), address, zip code, credit card number, social security number.
- *Anthropometric/biopsychosocial*: age (birthdate), height, weight, gender.
- *Medical*: medical history, medication, blood-related, mental health, smoking or alcohol use.

PI-based categorization. Mapping textual information attached to UI elements onto a PI semantics was nontrivial. We used techniques from Information Retrieval along with an iterative refinement process that aimed to increase precision (gradually reduce False Positives and False Negatives). Whereas Identity PI are relatively straightforward to find, other PIs require searching for adjacent terms. In Table 1 we show examples of such adjacent terms.

Table 1. Information retrieval: adjacent terms for PI extraction

PI	Adjacent terms
Medical history	Surgery, Allergy, …
Medication	Prescription, Dosage, Dose, Drug, …
Blood	Glucose, Cholesterol, Oxygen, Pressure, …
Mental health	Stress, Panic, Anxiety, Depress, …

2.2 Leak Analysis

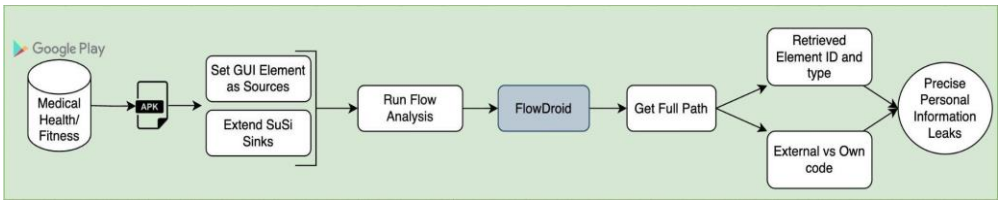

Figure 3. Overview of PIT's leak (information flow) tracking process

Flow analysis. Figure 3 shows the pipeline PIT uses to detect PI leaks: it employs a flow analysis – a standard program analysis that tracks flow of data from a *source* (origin) to a *sink* (destination) – to find how sensitive information collected by the app can leak to untrusted destinations. We define a *leak* as a path from a source to a sink; when multiple paths exist from

one source to one sink, we only count the shortest path. Prior work on security has typically set trusted device identifiers as sources, and untrusted destinations (network, SMS) as sinks. However, in our approach, we set UI components as *sources*, while as *sinks*, we set any method that can potentially leak user information, e.g., methods that send data to the network, or write to local storage. To create the list of sink methods, we started with the SuSi list (widely-used in security research (SuSi, 2024)) as our baseline and expanded it by manually adding more method signatures, tripling the count of the baseline. As flow analyzer, we used the standard FlowDroid tool (Arzt et al., 2014), which requires specifying Java methods as sinks and sources. However, in Android, UI elements are specified as XML objects, not as Java methods.

In our previous work (Ardalani et al., 2024), we used the JADX tool to map object identifiers in XML to their Java creation code in the Android-specific *R.java* file, and from there we set the *findViewById* method as a source (*findViewById* is used to connect a UI element to its corresponding code section). However, using *findViewById* may lead to false positives, because *findViewById* can be used to extract (or connect to) UI object *properties* like font size, color, or the position of a view are tracked, rather than the contents of the UI object (e.g., text that contains PI). Therefore, we refined our analysis with a new approach to reduce these false positives. Figure 4 illustrates the overall enhanced process used to identify and set view sources for flow analysis, while Figure 5 presents a code example that demonstrates this process in action. The first step is a manual investigation of methods responsible for retrieving data from *View* elements, such as *getText()* for *EditText* view objects. We identify a collection of approximately 100 such methods and use FlowDroid to analyze the data flow from these methods to extended sink methods. In Figure 5, line 6 serves as the source (*editTextInput.getText()*), and line 7 is the sink (*fileOutputStream.write(text.getBytes()*), where the user input flows to the file.

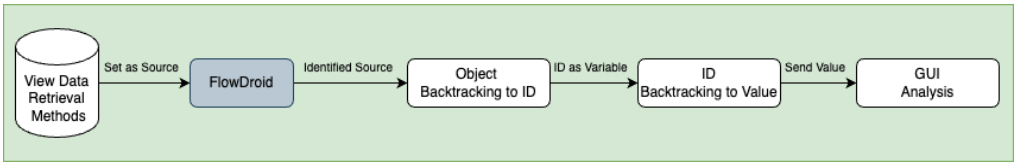

Figure 4. Overview of configuring *View* elements as sources for information flow analysis

The next stage of the analysis is **Object Backtracking to ID**. Here, we track the view object (`editTextInput`), on which the source method is invoked, to locate the corresponding *findViewById()* call and identify the ID associated with it. In our example, tracking `editTextInput` leads us to line 3, where the ID associated with it is `IdOfView`. In some cases, the ID is a constant value, but more often, the ID is an integer variable. Therefore, a further analysis step, depicted in Figure 4 as **ID Backtracking to Value**, is required to determine the value of the variable. In our example, the goal is to track `IdOfView` back to its value. As illustrated, the ID originates from the return value of the method `integerID()`, which is defined in line 10 and returns the constant value 100. Finally, after obtaining the constant integer value, we conduct a **GUI Analysis** (as shown in Figure 1), which involves mapping the integer ID back to the corresponding entry in the *R.java* class. This class serves as a bridge between Java code and the layout XML file, allowing us to link the integer ID to the appropriate UI element.

```
1   protected void onCreate(Bundle savedInstanceState) {
2       int IdOfView = integerID();
        ...
3       editText = findViewById(IdOfView);
        ...
4       saveName(editText);
        ...
    }

5    private void saveName(EditText editTextInput) {
6       String text = editTextInput.getText().toString();
        ...
7       fileOutputStream.write(text.getBytes());
        ...
    }

8    private integerID(){
9       return 100;
     }
}
```

Figure 5. Example code demonstrating the flow of user input from a *View* element (`getText()`) to a sink (`write()`).

Own-code vs. external-code flows. Another crucial aspect of precise leak attribution is distinguishing between own-code and external-code. Own-code is primarily identified by the application package name. For example, in app *com.gotokeep.yoga.intl.apk*, all the code whose package is *com.gotokeep.** or *com.gotokeep.yoga.** is considered own-code, whereas code in packages *io.branch.** or *com.facebook.** is considered external-code. Of course, code in package *com.facebook.** would be considered own-code in the Facebook app itself.

3. RESULTS

App dataset. We ran our approach on 2,832 Android apps: 2,211 from Google Play's Medical category and 621 from Google Play's Health&Fitness category. We used two criteria for including apps in our analysis: (1) a minimum popularity threshold (≧ 500 installs), and (2) the apps had to be in English.

3.1 What is the Prevalence of PI Collection?

Figure 6 shows the prevalence of PI collection. We found that *Email* is by far the most collected information, with 44% of Health&Fitness apps (`Fitness' for short), and 39% of medical apps collecting it, respectively. Next, we found that *Age, Name (first, last), Phone number, Address, Gender, Height*, are collected by about 15%–20% of the apps. As expected, the collection of specialized medical information (e.g., *current medications, medical history, use of smoking/alcohol*) is more prevalent in medical apps. However, we were surprised to see that fitness apps have a comparatively higher prevalence of collecting identity information such as *email, weight, age, gender, height*, compared to medical apps.

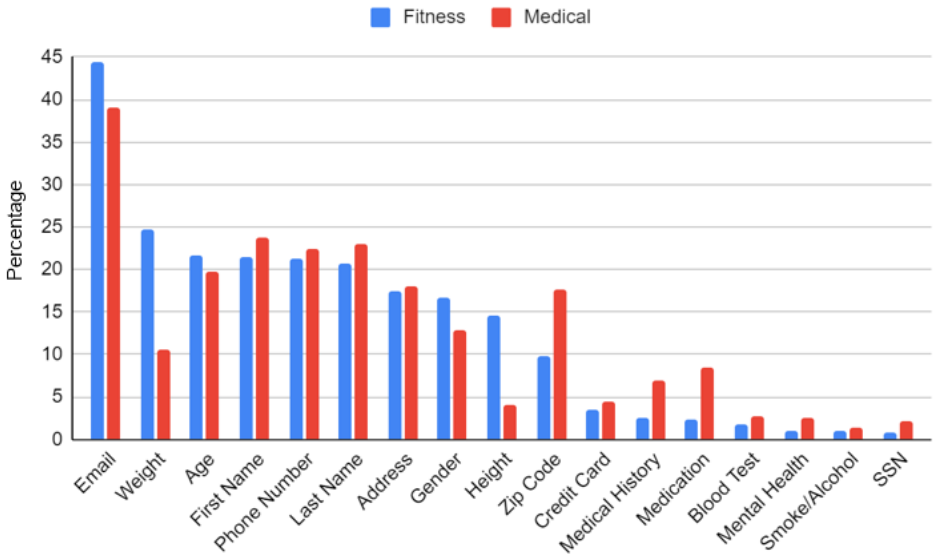

Figure 6. Prevalence of PI collected

3.2 Own-Code vs. External-Code Leaks

We begin with two motivating examples that illustrate the difference between own- and external-code leaks

Motivating example: Own-code leak. The mental health app *Panic Shield* (*com.panic.shield*) is designed to protect users from panic attacks. The list of user phobias, labeled fear8name, is among the PI it collects; the app collects this information via an *EditText* and stores this information in the local *SharedPreferences* database. The result of flow tracking, that starts at the source and ends at the sink, indicates the methods and VM registers involved in the flow:

SOURCE
$r1=virtualinvoker0.<com.panic.shield.exposure.CreateHierarchy:android.view.View findViewById(int)> ⇒
$r23 = r0.<com.panic.shield.exposure.CreateHierarchy: java.lang.String l> ⇒
Interfaceinvoke $r25.<android.content.SharedPreferences$Editor: android.content.SharedPreferences$Editor putString(java.lang.String,java.lang.String)>("fear8name", $r23) SINK

We consider this a *own-code leak* since all the code involved belongs to the app (or Android itself).

Motivating example: external-code leak. The *Keep Yoga* fitness app (*com.gotokeep.yoga.intl*) collects the user's gender (encoded as a boolean) and uses the *io.branch* external-code library, which will store the gender in the local *SharedPreferences* database. The result of flow tracking is:

SOURCE
r11 = virtualinvoke $r6.<android.app.Activity: android.view.View findViewById(int)>($i0) ⇒
$r1 = <io.branch.referral.PrefHelper: io.branch.referral.PrefHelper prefHelper_> ⇒
putBoolean(java.lang.String,boolean)>($r1, $z0) SINK

Note how in the flow from collecting the user's gender (source) to storing it in the database (sink), there is a method from the *io.branch external-code* library. This library is primarily used for deep linking and user engagement (Table 3). The presence of third-party methods, such as those from *io.branch*, makes this *external-code flow*. Note how external-code flows increase the risk of data leakage: the data, once handled by external-code, can be stored locally but also potentially shared with third-parties (Reardon et al., 2019). Therefore, this second leak scenario, involving the sharing of data with external-code, is potentially more dangerous due to the additional risk of exposing personal information to external entities.

Table 2. Leak statistics

	Own-code	External-code
Median	6	29
Average	43.29	156.34
Max	978	2559

Table 2 shows statistical measures for leaks: we found that the typical app has 6 own-code leaks and 29 external-code leaks (the average values are affected by apps that have a large number of leaks, up to 978 own-code, and 2,559 external-code, as can be seen in the last row). We believe that these results are concerning, especially from the standpoint of external-code leaks. *When the ratio of leaks induced by* external-code *libraries to an app's own code is 29:6, essentially the developer has long lost control over who collects user data, and how this data is processed or sent.*

Overall leaks. Figure 7 shows histograms for the overall number of leaks, as well as own-code vs external-code leaks. The histogram on the left shows that 158 apps have less than 20 leaks. However, please note that, to be included in the figure, an app had to have at least one leak. The histogram shows that dozens upon dozens of apps have a substantial number of leaks: 48 apps have 20–40 leaks, 25 apps have 40–60 leaks, and *128 apps have more than 100 leaks*. These numbers indicate that a high number of leaks is not an isolated incident; this provides an impetus to find and analyze apps with a high number of leaks.

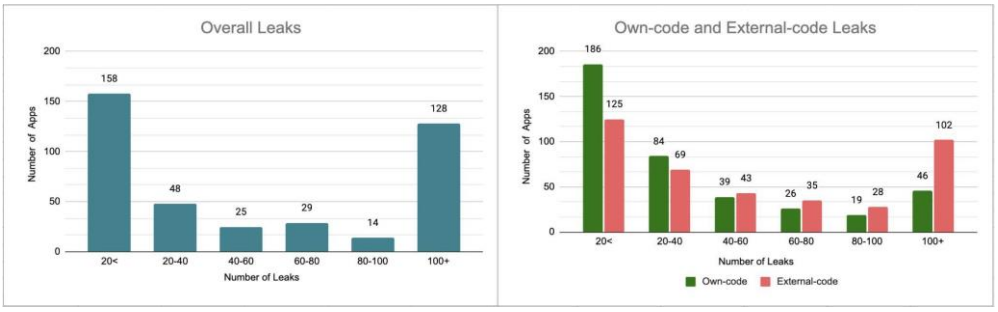

Figure 7. Leak distributions: overall (left) and own- vs -external code leaks (right)

Most prevalent external-code libraries. In Table 3 we show the nature and prevalence of top external-code libraries. For example *com.facebook* (which implements analytics and social media integration services) appears in 87 apps; *io.fabric*, which provides analytics, crash reporting, and user engagement services, appears in 38 apps. Note that an app may contain more than one of these external-code libraries, and there is a lot of fragmentation. While, for space reasons, we only present the top-9 libraries, there are dozens of other external-code libraries that we found in apps.

Table 3. Most frequent external-code libraries and their nature

	#Apps	Ads	Analytics	Payment Processing	Crash reporting	User engagement	Deep linking	Authenti-cation	Social Media Integration
com.facebook	87		•						•
com.squareup	58		•	•					
io.fabric	44		•		•	•			
io.branch	38					•	•		
net.hockeyapp	34				•	•			
com.crashlytics	28		•		•				
com.appboy	26		•			•			
com.mopub	24	•							
com.applovin	20	•	•						

3.3 Where does the PI Leak?

We now discuss each leak destination, quantified in Table 4. *Net* indicates that the information is leaked over the network; these leaks are the most concerning, because the moment the information has left the phone, the user has de facto lost control over where the information propagates. Please note that characterizing the network *domain* where information leaks to over the network – e.g., the app's own servers, or analytics/advertisers – is challenging and requires a dynamic analysis (Wei et al., 2012). *Log* indicates that PI information is leaked to system logs; this is problematic because system logs are visible to any app. *Local DB/Bundles/Shared Pref* indicates that the information is leaked to local persistent storage (*Bundle* is the Android serialization mechanism, and *SharedPreferences* store user preferences). *File/IO* indicates that the PI is saved in local files. Note that leaks to local storage (DB, files) have been proven to be used by malicious apps and malicious libraries to exfiltrate data (Reardon et al., 2019). The table reveals that, depending on the destination, external-code leaks are 1.8x–11.9x more numerous than own-code leaks. For brevity we omit showing the distribution for each destination, however we found that apps that have own-code network leaks typically have few (<20), whereas 40 apps had ≥ 20 network leaks (they are all external-code leaks).

Table 4. Leak destinations

	#Leaks	% of overall leaks	Own-code	External-code
Net	11002	24.59	916	10086
Local DB/Bundle/SharedPrefs.	16595	37.08	3073	13522
Log	10325	23.07	3687	6638
File, I/O	6831	15.26	529	6302

3.4 Fine-Grained PI Leak Analysis

Table 5 shows fine-grained results: the number of apps in each category that exhibit at least one leak for that PI, as well as totals for each PI and each destination. Note that an app can have multiple leaks of the same PI to the same destination, e.g., the *Email* is saved into two different files, or sent onto the network via two different connections. *In this subsection only, when referring to "leaks" we count apps, not individual leaks; in other words, any app that has more than one PI→destination leak is only counted once.* Overall, Medical apps have about 3x more leaks than Fitness apps (1490 vs. 510) in part due to medical apps collecting specific medical PI; nevertheless, some Fitness apps still collect and leak medical PI such as medication, blood, or smoking/alcohol use. *Weight*, *Email*, *First Name*, and *Gender* are the most frequently leaked personal information in Fitness apps, while *Email*, *Credit Card*, *Phone Number*, and *Address* are the most commonly leaked in Medical apps.

Interestingly, while 44% of Fitness apps collect the *Email*, only about 14% leak it (see Figure 1); whereas 39% of Medical apps collect the *Email*, and 34% leak it. Another interesting point is that Fitness apps have a much higher tendency to collect and leak *Weight* and *Height*. Interestingly, Fitness apps show a greater tendency to collect and leak data related to physical attributes, such as *Weight* and *Height*, reflecting their focus on users' physical characteristics. On the other hand, *Medical* apps are more likely to collect *Credit Card* information, along with related personal information like *Addresses* and *Phone Number*, which are commonly used in payment processing. A positive finding is that *Social Security Number* (*SSN*) were not leaked in any of the analyzed applications. We also noticed that Medical apps have 2.5x more network leaks than Fitness apps do (341 vs. 133), making them higher risk (and inviting more scrutiny) than Fitness apps.

Table 5. Fine-grained PI leak information (#of apps exhibiting *one or more leaks* of that PI)

	Fitness					**Medical**				
	Net	Log	DB	FileI/O	*Total*	Net	Log	Net	FileI/O	*Total*
Email	24	25	25	15	*89*	85	64	84	54	*287*
First name	11	7	10	7	*35*	16	13	30	23	*82*
Last name	10	6	7	5	*28*	18	18	33	20	*89*
Phone	6	3	4	4	*17*	46	25	60	44	*175*
Address	5	12	11	4	*32*	33	35	78	30	*176*
Zip	5	3	4	3	*15*	11	11	14	8	*44*
Gender	8	8	13	5	*34*	15	15	18	7	*55*
SSN	0	0	0	0	*0*	0	0	0	0	*0*
CCard	9	8	8	5	*30*	52	33	92	52	*229*
Age	4	9	3	2	*18*	15	20	18	11	*64*
Weight	28	34	34	19	*115*	7	17	16	6	*46*
Height	8	10	9	4	*31*	4	8	11	4	*27*
Medical hist.	5	4	6	4	*19*	18	25	31	12	*86*
Medication	3	3	3	2	*11*	15	22	27	11	*75*
Blood	3	3	3	2	*11*	9	5	8	2	*24*
Mental health	3	7	8	2	*20*	6	8	10	3	*27*
Smoke/Alcohol	1	1	3	0	*5*	1	0	3	1	*5*
Total	*133*	*143*	*151*	*83*	*510*	*341*	*319*	*533*	*288*	*1490*

3.5 Frequency of Personal Information Leaks Across Apps

In this section, we examine the occurrence of personal information (PI) leaks across various apps. Figure 8 illustrates the distribution of PI leaks by frequency. Notably, approximately 50% of the apps exhibit a single PI leak, with the most commonly leaked PIs being Email, Medical History, and Address. Around 22% of the apps leak two PIs, with the combinations (Address, Credit Card), (Height, Weight), and (First Name, Last Name) being the most frequent pairs. For apps leaking three PIs, the most prevalent groupings are (Credit Card, Email, Phone) and (Email, First Name, Last Name). In instances where apps leak between four and nine PIs, no consistent pattern emerges. The leaked information varies significantly from one app to another, indicating less predictability in the types of information exposed when multiple PIs are involved.

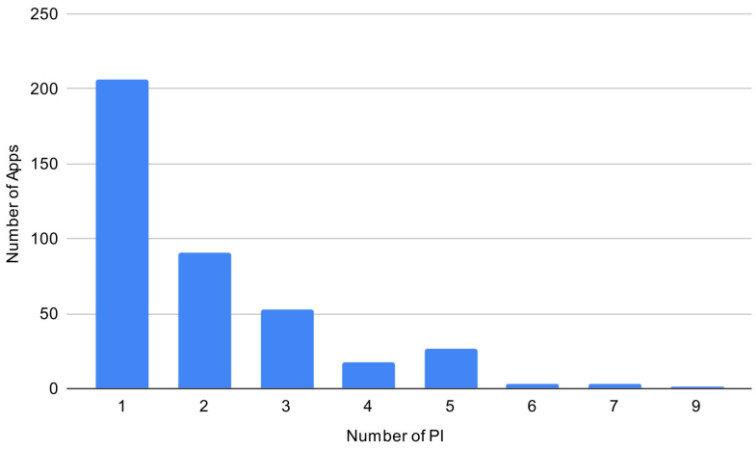

Figure 8. Frequency distribution of personal information (PI) leaks by the number of PIs leaked per app

Figure 9 illustrates the presence of each pair of leaked personal information (PI) across the apps, showing how frequently these PI pairs are exposed. Notably, Credit Card appears in three out of the five most frequently leaked pairs, occurring in more than 40 apps. The PIs associated with Credit Card, such as Email or Phone Number, likely play a role in payment processing, serving as contact information for payment notifications or identification. Additionally, First and Last Name are commonly leaked alongside Credit Card, reflecting their relevance as the cardholder's name. Address is often included as well, typically for billing or shipping purposes. In 43 apps, both First and Last Name were leaked together, which is unsurprising. Furthermore, Email, either independently or in combination with these names, was leaked in 30 apps. The remaining leaked PI pairs also offer valuable insights and warrant further investigation.

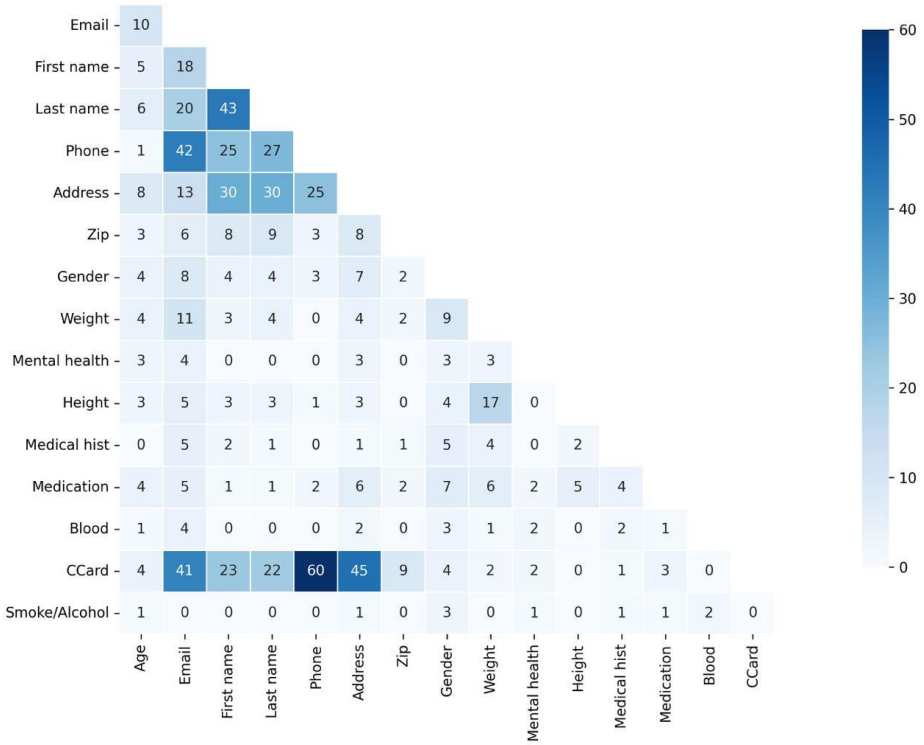

Figure 9. Frequency of PI pairs across apps

3.6 Characterizing Network Links

Section 3.3 and Table 4 have indicated that 24.59% of the leaks were to the network. However, that does not indicate which entity the app is communicating with, over the network. When apps send or receive data, knowing the destination (or origin) of this data helps us determine the nature of the app, the nature of the communication, and potential areas of concern. Furthermore, it can also help the end-user track what websites or services are receiving their data or help prevent misuse of that data. To address this issue we analyzed *links*, specifically the URLs embedded in apps, and then devised a characterization of these links.

Extracting links. To extract links, we first decompiled the app using the JADX (JADX, 2024) and apktool (Apktool, 2024) decompilers, extracted all the URLs found in the decompiled version, and finally categorized the links. When classifying links, we grouped them into one of four classifications: first-party, third-party, advertising, and Google APIs. Our data, displayed in Table 6, shows that Google APIs are the most popular links (46.51%), followed by third-party links (37.64), then first-party links (11.79%).

Table 6. Link categories and their prevalence

Statistic	First-party	Third-party	Advertising	Google APIs
Total	16872	53835	5786	66516
Percent	11.79	37.64	4.04	46.51

First Party. These links indicate communication between the app and its own servers. To identify first-party links, we matched the URL's domain name to a subsection of the app's package name; if there was a match, the link would be classified as first-party. For example, links to 'www.logbox.co.za' were considered first-party when accessed in app 'za.co.logbox'.

Google APIs. These links indicate the use of Google services, e.g., Firebase storage, `Sign-in with Google', etc. We identify such links based on the URL containing googleapis.com. Note that some Google API links could be advertising ('www.googleapis.com/auth/display_ads').

Advertising. We deemed links as advertising if they point to known advertising providers, i.e., the URLs contained the domains: 'amazon.com', 'facebook.com', 'aboutads.info', 'shoppable.com', 'supersonicads.com', 'googleadservices.com', 'kargo.com', 'appsfire.com', 'nativex.com' and 'tapestrylabs.com', etc.

Excluded Links. Apps contain an abundance of links to Web standards, and Web frameworks, e.g., 'www.w3.org' or government reference material ('www.<resource>.gov' or `www.<resource>.gov.>country>). We excluded these links from the analysis as we considered them to be routine/innocuous – simply an artifact of the apps using the Web. Moreover, when apps connect to these URLs, they generally do not send user data.

Most Prevalent Link classifications. Figure 10 shows the number of links found in each app. The average app has 60.52 embedded links across all classifications; the geometric mean for all classifications was 25.12. Overall, out of 2,831 apps studied, 468 apps had no categorizable/valid links, 318 had 1–20 links, 704 had 20–40 links, 229 had more than 100 links, and 9 had more than 1000; these figures illustrate the breadth of app communication and the variety of URLs apps exchange data with.

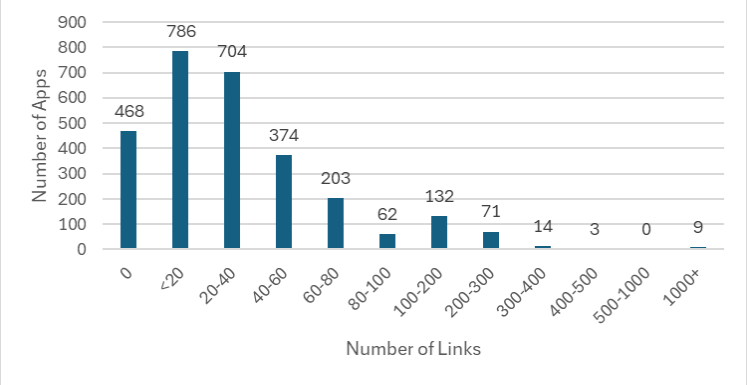

Figure 10. Links per app distribution

Most link-heavy apps. Table 7 shows the top-ten apps with the most links for each classification. The app with the highest number of overall links was org.iggymedia.periodtracker, with *9574 third-party links, and 168 known advertising links.* The most common domains in this app were webmd.com, babycenter.com, and healthline.com. The app with the highest number of first-party links was webmd.com, with 8,598 first-party links, which we attribute to WebMD being a reference app. App infirmiers.pro had the highest number of advertising links (1498, which is *a quarter* of all advertising links in our 2832 app dataset).

Table 7. Top-10 apps with the most embedded links

Most Link-Heavy Apps & their Classifications					
APK Name	First-party	Third-party	Advertising	Google APIs	*Top 3 Most common Links*
org.iggymedia.periodtracker	0	9574	168	31	*webmd.com* *babycenter.com* *healthline.com*
com.webmd.android	8598	12	2	53	**.webmd.com* *googleapis.com* *www.idangero.us*
com.WegileWildCard.transform	0	5744	1	53	*dropbox.com* *googleapis.com* *gravatar.com*
infirmiers.pro	0	514	1498	31	*facebook.com* *googletagmanager.com* *googleapis.com*
com.phillips.cdp.ohc.tuscany	392	1129	1	17	**.philips.** *nakupovanje.net* *googleapis.com*
com.lf.lfvtandroid	0	1472	1	45	*youtube.com* *googleapis.com* *lifefitness.com*
gov.va.general.med.ee	1287	20	0	22	**.va.gov* *googleapis.com* *nap.edu*
wikem.chris	0	1257	18	18	*youtube.com* *mdcalc.com* *thepocusatlas.com*
uk.co.classprofessional.cpg	18	1031	0	1	**.community.librios.com* *evidence.nhs.uk* *gravatar.com*
com.bracemateapp.bracemate	29	623	0	1	*dropbox.com* *google.com* *youtube.com*

Table 8 shows that a typical Medical app communicates with 27 URLs, whereas a typical Health&Fitness app communicates with 39 URLs. Table 9 shows statistics on links, separated by their classifications. A typical Medical app has 3 first-party links, 7 third-party links, 2 advertising links, and 24 Google API links. In contrast, a typical Health&Fitness app has 4 first-party links, 9 third-party links, 3 advertising links, and 29 Google API links. This data suggests that Health&Fitness apps have a higher inclination to communicate data to the URLs.

Table 8. Statistics by app Category

Statistic	Medical	Health And Fitness
Max	2043	9773
Average	41.68	124.91
Geometric Mean	21.7	41.57
Median	27	39

Table 9. Statistics by link category

Health And Fitness Apps				
Statistic	First-party	Third-party	Advertising	Google APIs
Max	8598	9574	168	234
Average	103.75	53.63	4.34	59.34
Geometric Mean	5.47	9.41	2.71	28.48
Median	4	9	3	29.5
Medical Apps				
Statistic	First-party	Third-party	Advertising	Google APIs
Max	1288	1257	1498	231
Average	17.84	15.14	4.06	29.47
Geometric Mean	3.98	6.26	2.16	20.19
Median	3	7	2	24

3.7 Characterizing the UI elements that collect PI

In total, our apps contain 47,749 Android *View* objects (i.e., GUI elements) that collect PI. Figure 11 shows the *View* types, along with the semantics of the PI they collect. For each type, we also indicate the top-3 most frequent PI that is collected with that *View*. The most prevalent UI object is *EditText* (50.8%) which allows arbitrary text input. Naturally, *EditText* is used to collect emails, addresses, first names, and so on. However, the flexibility of *EditText* can be a downside as well, when the information collected needs to have a certain type (e.g., numeric) or range, e.g., 0–100; in such cases, developers need to add input validation, whereas other controls, e.g., *Spinner* or *SeekBar*, can directly enforce a certain discipline on values or ranges. *RadioButton* and *CheckBox* are tied for the second most frequent PI collectors, typically used to select the gender, an age range, or a list of medications/medical conditions.

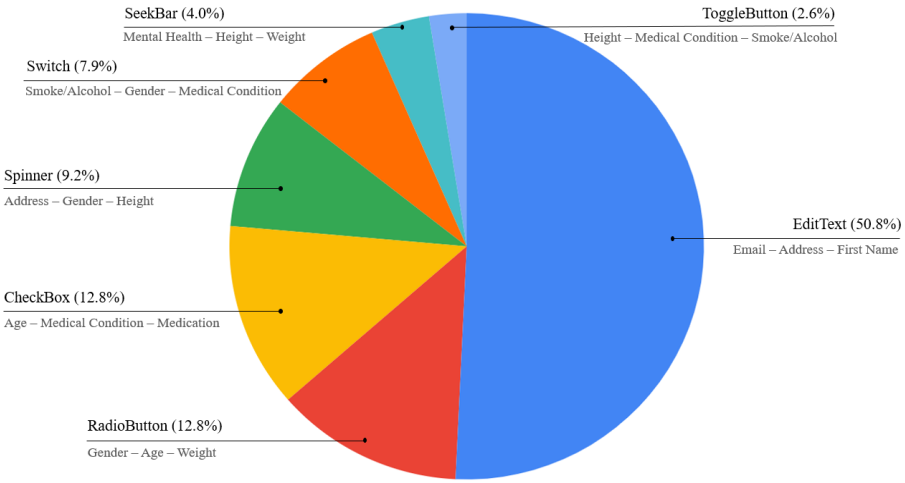

Figure 11. UI element distributions and the PI they collect

3.8 Discussion

We believe that (1) such apps should move toward collecting and leaking less, and (2) users, developers, app markets, and regulators can all play a role. First, users should question developers' over-collection, e.g., an app that simply computes the BMI should just ask for weight and height, and not ask for the user's medical history. Second, as developers typically use third-party code for monetization (ads), they should balance revenue with the leaks that third-party code induces; similarly, users can put pressure on developers to reduce ads and leaks. Third, app markets can be more transparent about the data apps collect, e.g., offer a detailed description of the data collected, and where this data is stored/sent. Fourth, regulators should be much more aggressive in enacting measures to make apps transparent about collection, and protective of user data.

4. TOOL: PERSONAL INFORMATION TRACKER

Our approach is implemented in a tool called **PIT (**Personal Information Tracker**)**, which, along with its documentation, is publicly available on our GitHub repository at github.com/Alireza-Ardalani/PIT. By default, PIT supports 17 types of personal information as sources and tracks 4 types of sinks: logs, network, database (DB), and input/output (IO) operations. However, PIT can be customized to accommodate other types of sources and sinks through a configurable setup. Figure 12 illustrates the output generated by the PIT tool when applied to the com.phr.PayNow application. In this representation, each type of personal information is shown as a label inside a square, while each sink type is depicted within a circle. The edges represent the flow of personal information to its corresponding sink, with red edges indicating external-code data flows and green edges representing own-code flows.

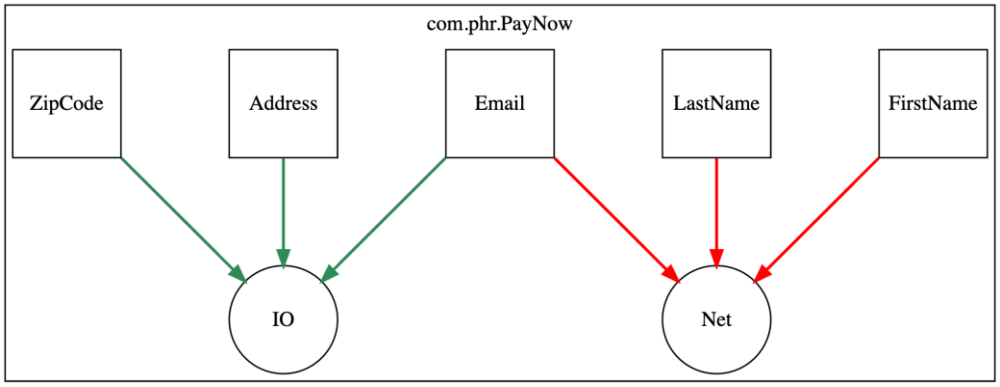

Figure 12. Output of PIT for com.phr.PayNow app

5. CONCLUSIONS AND FUTURE WORK

Our study has revealed that Medical, as well as Health&Fitness apps, collect and leak a plethora of personal information. We believe that our work could be extended along several directions. First, a dynamic analysis that captures the destination of PI would provide a more precise temporal dimension of when data is collected, and how often it is transmitted to the network. Second, our current toolchain runs locally; we envision it could be extended to collect and report data for a given individual app to the end-users as a browser extension or directly on the phone. Finally, our analysis could be combined with a policy analysis to determine (and inform users) of app compliance with privacy regulations such as the GDPR in the EU or CCPA in California.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CCF-2106710.

REFERENCES

Ardalani, A., Antonucci, J. and Neamtiu, I., 2024. Towards Precise Detection of Personal Information Leaks in Mobile Health Apps. 18th *IADIS International Conference on e-Health (part of MCCIS 2024)*. Budapest, Hungary.

Arzt, S., et al., 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*, Vol. 49, No. 6.

Apktool, 2024. *A tool for reverse engineering Android APK files*. Available at: https://ibotpeaches.github.io/Apktool/

Continella, A., et al., 2017. *Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis* (Report for NDSS Symposium 2017). Available at: https://publik.tuwien.ac.at/files/publik_278933.pdf

HIPAA, 2024. *US Department of Health & Human Services: Guidance Regarding Methods for De-identification of Protected Health Information in Accordance with the Health Insurance Portability and Accountability Act (HIPAA) Privacy Rule*. Available at: https://www.hhs.gov/hipaa/for-professionals/privacy/special-topics/de-identification/index.html#protected

Huang, J., et al., 2015. SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps. *Proceedings of the 24th USENIX Security Symposium (SEC'15)*. USENIX Association, USA, pp. 977-992.

JADX, 2024. *Dex to Java Decompiler*. Available at: https://github.com/skylot/jadx

Jia, Q., et al., 2019. Who Leaks My Privacy: Towards Automatic and Association Detection with GDPR Compliance. In E. Biagioni, Y. Zheng and S. Cheng (eds.) *Wireless Algorithms, Systems, and Applications. WASA 2019. Lecture Notes in Computer Science*, vol 11604. Springer, Cham. https://doi.org/10.1007/978-3-030-23597-0_11

McClurg, J., Friedman, J. and Ng, W., 2013. *Android Privacy Leak Detection via Dynamic Taint Analysis*. Available at: https://jrmcclurg.com/papers/internet_security_final_report.pdf

Rahaman, S., Neamtiu, I. and Yin, X., 2021. Algebraic-datatype Taint Tracking, with Applications to Understanding Android Identifier Leaks. *ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens, Greece, pp. 70-82.

Reardon, J., et al., 2019. 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System. *Proceedings of the 28th USENIX Security Symposium*, pp. 603-620. Available at: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon

Ren, J., et al., 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. *MobiSys '16: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 361-374. Available at: https://dl.acm.org/doi/abs/10.1145/2906388.2906392

Samat, S., 2022. *Living in a Multi-device World with Android*. Available at: https://blog.google/products/android/io22-multideviceworld/

SuSi, 2024. Available at: https://github.com/secure-software-engineering/SuSi

Sweeney, L., 2000. *Simple Demographics Often Identify People Uniquely* (Data Privacy Working Paper 3). Available at: https://dataprivacylab.org/projects/identifiability/paper1.pdf

Van Alstin, C., 2024. Massive Data Trove from Change Healthcare Hack Now for Sale on Dark Web, *Health Exec*. Available at: https://healthexec.com/topics/health-it/cybersecurity/massive-data-trove-change-healthcare-hack-now-sale-dark-web

Wei, X., et al., 2012. ProfileDroid: Multi-layer Profiling of Android Applications. *Mobicom '12: Proceedings of the 18th annual international conference on Mobile computing and networking*, pp. 137-148.