# What's Inside My App?: Understanding Feature Redundancy in Mobile Apps

Yao Guo, Yuanchun Li, Ziyue Yang, Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
{yaoguo,liyuanchun,yzydzyx,cherry}@pku.edu.cn

## ABSTRACT

As the number of mobile apps increases rapidly, many users may install dozens of, or even hundreds of, apps on a single smartphone. However, many apps on the same phone may contain similar or even the same feature, resulting in *feature redundancy*. For example, multiple apps may check weather forecast for the user periodically. Feature redundancy may cause many undesirable side-effects such as consuming extra CPU resources and network traffic. This paper proposes a method to identify common features within an app, and evaluated it on over four thousand popular apps. Experiments on a list of apps installed on actual smartphones show that the extent of feature redundancy is very high. We found that more than 85% of user smartphones contain redundant features, while in extreme cases, some smartphones may contain dozens of apps with the same feature. In addition, our user surveys found out that about half of the redundant features are undesirable from the end users' perspective, which indicates that feature redundancy has become an important issue that needs to be investigated further.

## KEYWORDS

Mobile apps; feature; app bloat; redundancy; Android.

**ACM Reference Format:**
Yao Guo, Yuanchun Li, Ziyue Yang, Xiangqun Chen. 2018. What's Inside My App?: Understanding Feature Redundancy in Mobile Apps. In *ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension, May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196329

## 1 INTRODUCTION

With the prevalence of smartphones, new mobile applications (*apps* for short) have been developed and released at a dramatic speed. The number of apps in both Apple App Store and Google Play has surpassed the two million mark in 2016, while the number of apps in Google Play has reached three million in 2017. The total number of app downloads in Google Play and App Store worldwide was at 27 billion in the final quarter of 2017, according to App Annie.

With so many apps available for downloading, and most of them free, most smartphone users have installed dozens, or even hundreds of, apps on their smartphones. According to the Yahoo Aviate study in 2014, Android users have an average of 95 apps installed on their phones [29].

At the same time, many apps tend to include more and more features in a single app. A popular weather app may evolve into an "all-around" app with all kinds of features including news feed, app management, schedule management, map and navigation, etc. This phenomenon has sometimes been designated with the term *software bloat* or "bloatware".

As there are many apps installed on a smartphone and some of these apps may contain features not designed as their main feature or functionality, it becomes inevitable that some of the features in these apps may become redundant. For instance, a user may install multiple apps that check weather forecast periodically on his/her smartphone. In this paper, we refer to this phenomenon as *feature redundancy*.

Feature redundancy not only takes up valuable memory and space on the phone, it may also cause other undesired side-effects such as wasted CPU, network traffic and extra battery consumption. Having more than one apps checking weather forecast information for the same user on the same smartphone is potentially a waste of network data traffic and battery consumption. The situation becomes worse if there are not only a few, but more than a dozen apps, doing the same things over and over again.

The goal of this paper to *identify redundant features from the installed apps and understand the extent of feature redundancy on a smartphone.* We first present a simple NLP-based method to identify common features from an Android app, and then evaluate the method on a set of popular apps. Once we successfully uncover the features within each app, we can study the extent of feature redundancy with a collection of used/installed apps from real smartphones.

Our first challenge is how to uncover the features provided by each app. As many features are not advertised in an app's description, we cannot trust feature identification based on app descriptions. Thus, we propose a feature identification method based on the UI resources in each app's installation package. We use a simple NLP-based method to form a list of keywords related to each feature, and developed a taxonomy of six common features with their corresponding


*ICPC '18, May 27–28, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196329

keywords. With this taxonomy, feature identification has been transformed into a keyword-matching problem.

We evaluated the feature identification algorithm on a set of popular apps downloaded from both Google Play and a popular third-party market. Out of the 4,000 apps, we have identified more than 3,200 instances of the six popular features we studied. About 22% of the studied apps contain more than one potentially redundant features. Manual examination on a list of 50 top apps shows that the algorithm achieves a precision of almost 90%.

In order to evaluate the extent of feature redundancy, we use two sets of real-world user data: a large-scale used app list of 600,000 users from a popular app market, and an installed app list collected from 87 volunteers. When considering the large-scale dataset with used apps, we found that more than 85% of smartphones contain potentially redundant features, while 71% of them contain more than one types of potentially redundant features. When we consider the small-scale dataset with installed apps, the extent of feature redundancy becomes more severe, as 86 out of the 87 users contain some kinds of potentially redundant features, while more than 60% of these smartphones include all six redundant features.

We also designed a user study to ask users to indicate which features are unnecessary for each app. The survey result shows that 45% of the features surveyed are not desired from end users' perspective. The most undesirable features are schedule, app management, and email/messaging services.

Many users do not realize what kind of features are contained in each app, thus do not understand the severity of feature/function redundancy. Once they are presented with the information on potentially redundant features on their smartphones, many of them are surprised and want to know what they can do to reduce the level of redundancy. Thus we also discuss possible mitigation methods to remove/reduce some of the redundant features.

This paper makes the following key contributions:

- To the best of our knowledge, this is the first work revealing and demonstrating the prevalence of the problem of *feature redundancy* in mobile apps installed on Android smartphones.
- We propose a method to identify common features within an Android app. The method has been evaluated on more than 4,000 popular apps, with an identification accuracy of about 90%.
- We conduct a study on the extent of redundancy in real user data. The study shows that feature redundancy widely exists in the real world, with about 85% of the users having some kind of potential feature redundancy on the apps installed on their phones.

# 2 BACKGROUND AND RELATED WORK

## 2.1 Android Apps

Android has become the most widely used mobile platform for smartphones. At the same time, Android apps have been developed at a rapid speed. The number of Android apps has been increased much faster in recent year: from one million to two million in almost three years, while from two million to three million in a little over one year [28].

Users install apps to use various features, while the features are programmed by app developers. Some features are user-driven, which are activated upon user interaction and deactivated once the session ends, such as "making a phone call", "sending a text message" and "searching the web". Other features are not driven by user interaction, instead they periodically run in the background and notify users if necessary, for example a weather feature needs to continuously fetch weather data and tells the user about weather changes in (almost) realtime.

Users interact with Android apps through user interfaces (UI). The UI of an Android app reflects the features in most cases. In particular, the text found in UI often represents the features in an app. For example if there are words like "forecast", "sunny" and "temperature" in the UI, it may indicate that the app owns a "weather" feature.

## 2.2 App Bloat

Software bloat is a process where successive versions of a computer program become perceptibly slower, use more memory, disk space or processing power than the previous versions [34]. It is originally a problem found in Windows PCs, and now become an increasingly important problem in mobile devices as well [27].

Modern mobile systems are designed to prevent bloat, as it allows apps to share data with each other easily (for example using Intent [8] in Android). Mobile apps should be lightweight and streamlined for its own tasks, while ideally many small apps work together to complete complex tasks.

However, the software bloat problem also exists in the mobile platforms. Due to security concerns or financial considerations, many mobile app developers are adding more and more features to their own app, instead of making use of the existing (same) features in other apps.

App bloat is one of the main causes of feature redundancy. For example, app updates should be controlled by Google Play [33], so that only Google Play needs to check for updates of all the installed apps in the background. However, in order to be compatible with devices without Google services, many apps choose to check their update schedule separately. Thus it is quite common that many installed apps on a smartphone having a redundant "app management" feature, which means that they may waste resources to check individually whether their apps have a new version to download.

## 2.3 Feature Location/Extraction

Inferring and understanding the feature of code/software has been a popular research topic in the software engineering research community for a long time.

*Feature location*, which is known as the process of identifying an initial location in the source code that corresponds to a specific functionality, has been one of the most important and common activities performed by

programmers during software maintenance and evolution [9]. There are mainly three types of analysis methods for feature location: dynamic, static and textual analysis.

Dynamic analysis examines the code traces that are actually executed at runtime, and then binds them with features observed. The code traces can be compared to each other in order to find feature-specific code [10, 35]. Static analysis examines the internal structure of programs like control or data flow dependencies. This is typically how feature locations are manually done: programmers jumps between files according to module dependencies [4]. Textual analysis infers the feature of code pieces by the naming information. Common machine learning and information retrieval techniques are used in textual analysis.

Approaches targeting traditional software mainly make use of text processing techniques such as LSI (Latent Semantic Indexing) and LDA (Latent Dirichlet Allocation) to determine the features in a software based on the textual information in code. For example, Mudablue [16] and Tian *et al.* [30] use LSI and LDA, respectively, to automatically categorize the open-source repositories, regarding a software system as a document and an identifier as a word. Similarly, Kuhn *et al.* [17], Baldi *et al.* [2] and Maskeri *et al.* [23] proposed text-based methods to extract topics from code.

However, in modern software such as Android apps, textual information in code is often obfuscated in released versions. Researchers have explored other methods to infer the features within Android apps. Kanda *et al.* [15] extract features from source code of Android apps based on the pattern of API usage; WHYPER [24] makes use of app descriptions in market to infer the purpose of permission uses; Lin *et al.* [21] introduced the idea of inferring the purpose of permission by analyzing what third-party libraries an app uses; DroidJust [6] categorized the functionality of apps to different states based on the output events sensed by users.

Another similar line of work is the detection of cloned/repackaged Android apps, such as DroidMOSS [37], DNADroid [7], or WuKong [32]. However, they are typically more coarse-grained than feature identification, as they mostly focus on detecting the similarity of whole apps, instead of components within them.

In contrast to these related work, we detect the features of Android apps based on the textual information extracted from user interfaces, which reflects the app's actual behavior and cannot be obfuscated.

## 2.4 Redundant Operations on Devices

Redundant features often lead to redundant operations on smartphones, especially for the features with regular background tasks. Taking app management for an example, if every app checks their update schedule separately, it is a big waste of resource and energy. Apple requires all apps to be downloaded and updated though its official App Store. While Google recently also changed its policy to require all apps uploaded to Google Play update themselves through Google Play, many apps still prefer managing updates themselves.

Research has shown that redundant background tasks could waste battery [1, 5, 22, 31], produce unwanted network traffic [25] and leak users' privacy [13, 36]. Although there are a set of tools that can help kill the background tasks (task killers), the killed background processes may restart again and again by listening to some certain system events [11], thus even more resources might be consumed.

Many approaches have been proposed to reduce the resource consumption of redundant features. Some of them [3, 5, 12, 31] focus on optimizing background network traffic, such as grouping network traffic to save power. TAMER [22] controls the background activities that hold resource unnecessarily by monitoring, filtering, and rate-limiting. DefDroid [14] considers the continuous background behaviors in general as Disruptive App Behavior (DAB) and implemented a system to control the behaviors. ZipDroid [26] identifies the infrequently-used apps and disable them automatically to save smartphone resources. Li et al. [19] introduces a method to instrument mobile apps automatically to save sensor-related redundant behaviors.

We will talk about mitigating methods in Section 6, where some of these techniques will be useful.

# 3 FEATURE IDENTIFICATION IN MOBILE APPS

## 3.1 A Taxonomy of Features

Although feature location/identfication has been studied extensively in the community, there is no well-established definition on exactly what is a feature, especially when we need to determine the granularity of a feature. In this paper, we do not attempt to solve this issue and give an exact definition of feature. Instead, we define a set of criteria based intuition and heuristics, which will be used when deciding the set of features we will study in this paper. Specifically, we chose features based on the following criteria:

- Features in our study must be widely used features such as weather or app management, such that these features are needed on most smartphones.
- These features may be included in apps that are not designed for its main purposes. For example, although many apps are not originally designed for weather, they may include weather as an extra feature. Typical apps fitting into this description include browsers, map services, travel services, etc.
- The features may incur overhead or cost in resources or battery if they are redundant. Take weather as an example, multiple apps checking weather information periodically will incur undesirable side-effects.

Based on these criteria, we surveyed a list of top apps and selected the following six common feature:

- **Map/location**: providing mapping, navigation and location-based services;
- **Schedule**: providing calendar, reminder, event management services;

**Table 1: A taxonomy of features and keywords.**

| Feature | Keywords |
|---|---|
| map/location | topo locu mapquest googlemap topograph placemark zoom locat map |
| schedule | appoint calend remind alarm event agenda allday weekday calendar schedule |
| app management | uninstal download upgrad reinstal apk manage update install |
| weather | forecast timeweath accuweath foggi rain weather |
| news | feed newsfe hottest articl newslett news |
| email/messaging | mailbox sent unread draft sender mail trash inbox spam outbox |

- **App Management**: providing app search, download, update, installation service, including the update of the app itself;
- **Weather**: checking and displaying weather forecast information;
- **Email/messaging services**: sending and receiving emails/messages;
- **News**: checking and displaying news articles or feeds.

All these six features satisfy the requirements we mentioned above. They are common features that are needed by most users, while their functionality requires checking remote web services regularly or performing periodical operations. If any of these features are redundant on a smartphone, they may cause potential waste of valuable resources and battery.

## 3.2 Feature Identification Method

Identifying features and functionalities within an app is a difficult task, even with source code available. We face more challenges when identifying features from Android apps because many names are obfuscated in a majority of apps, thus it is difficult to gain enough information through analyzing the (de-compiled) code directly. As a result, although we have tried different methods in our attempt, including static code analysis and machine learning on string sets, as well as dynamic techniques involving screenshots, none of these approaches can predict the features within Android apps accurately.

In this paper, we propose a keyword-based method to identify the features in Android apps. Our method is based on the insight that the user interface (UI) of an Android app often reflects the features within the app [18], particularly the UI text, which is closely related to the features. Moreover, unlike the identifier names in code which could be easily obfuscated, the UI text, which will be presented to end users, cannot be obfuscated. Thus our goal is to uncover the features of an app by matching the keywords related to each feature.

*3.2.1 Main Idea.* The main idea of our method is considering each app as a document with its UI text as the content. Thus the problem of feature identification becomes similar to extracting topics from documents.

Given an app, if a feature is included in the app, there must exist some keywords related to that feature. For example, if an app provides the "Weather" feature, there must exist some keywords like "forecast", "foggy" and "rain", etc., which might be used to display the weather information. Whether a feature is included in an app can be determined according to whether there are enough feature-related keywords appearing in the app's UI text.

Our method can be summarized in three steps:

(1) *Extracting the UI text.* We make use of existing reverse-engineering tools to decompile Android apps and extract the UI text. With the UI text, each app is converted to a text document.
(2) *Generating a keyword list for each specific feature.* For each feature, we manually pick one or two keywords intuitively, and expand to more keywords based on the UI text extracted from a large set of apps in step 1.
(3) *Identifying the features within apps.* With the list of keywords for each feature generated in step 2, we identify the features of an app by matching the keyword list of each feature. A feature is identified in the app if we can find enough related keywords in the app.

The process of feature identification is shown in Figure 1. We first learn the keywords for each feature through a list of apps downloaded from Google Play. After keyword sets are collected, we identify features from each app by matching the words in its UI text with the keywords we have compiled. Next we will describe each of the steps in detail.

*3.2.2 UI Text Extraction.* The UI of an Android app is mainly programmed with the *XML* format under the *layout* directory. The strings used in the layout *XML*, which are the content of views in the UI, are stored in the resource files of the APK, and also represented in the *XML* format.

We make use of an existing reverse-engineering tool apktool[1] to obtain the resource files, and extract the key-value pairs from the *res/values/string.xml* file, which is the source of the app's word set. The keys of the pairs are the definitions of each string, which represent the developers' perspective of the UI. The values will be presented in the actual UI, which are what users will see. Both the keys and values are useful text resources used in our approach.

*3.2.3 Keyword Generation.* We use word2vec[2] in our keyword generation phase. Given some text data as a set of ordered word sequences, word2vec generates an N-dimension vector for every single word in the text data, where the vectors of words having similar meanings remain closer in the N-dimension space. Based on the vector set, we build a most-similar keyword set starting from one or two words.

Similarly, we can generate a keyword set for each specific feature. First we extract the string key-value pairs from the decoded files *(res/values/string.xml)* of over 13,000 apk

[1] https://github.com/iBotPeaches/Apktool
[2] https://github.com/danielfrg/word2vec

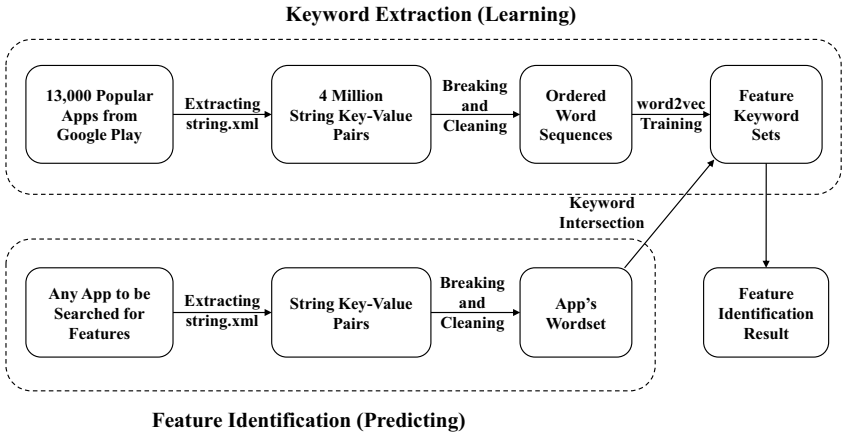



**Figure 1: The process of feature identification.**

files from Google Play (which were crawled during our previous work [18]) as the set of ordered word sequences. Next, we perform some cleaning to the sequence set, such as breaking down the identifiers with camel case and underscore-connected naming to a word list, filtering out stop-words and words with lower tf-idf, stemming, etc. Then we put the sequence set as the training data into word2vec.

Next, for each specific feature, we intuitively pick one or two words as the starting keyword set, then get a larger keyword set using word2vec. Finally, we manually removed some common words from the keyword sets because some of them hardly represent any meaningful features. The resulting keywords for each feature are listed in Table 1.

*3.2.4 Feature Identification.* To identify features in an app, we first extract a word set from the UI text, and perform the cleaning similar to the keyword generation phase. Then for each feature, we find the words appearing both in the feature's keyword set and the app's word set. A feature is found in an app only when the number of matched keywords is greater than a threshold, which is heuristically set as 3, based on our manual examination on some top apps.

## 3.3 Experimental Setup

We have downloaded a list of popular apps to evaluate the feature identification method. The apps are downloaded from two app stores: the official Google Play market and a popular third-party market CoolAPK[3]. We downloaded more than 2,000 apps from each market and use a list of around 4,000 apps in our study after identical apps are removed based on their package names.

The statistics of these apps are shown in Table 2. Note that during the evaluation process, we have used a different set of apps from the list of apps used to build the keyword list. Besides Google Play, we also included about 2,000 apps from a popular app market in China, because the users in our user study (Section 4) are mainly from China.

[3]http://coolapk.com

**Table 2: The number of apps used in our study.**

| App Market | # of Apps | Description |
|---|---|---|
| Google Play | 2127 | Most popular Android apps from different categories. |
| Cool APK | 2169 | Top downloaded Android apps based on market rankings. |
| **Total** | 4059 | *Note that duplicated apps from two stores are removed.* |

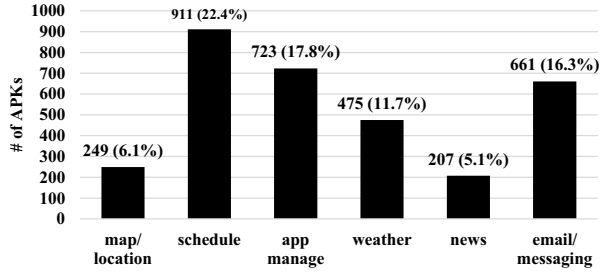



**Figure 2: The number of apps detected with each feature from 4,059 apps.**

## 3.4 Results and Analysis

We run the feature identification method on all 4,000 apps and get the list of features detected in each app. The results are shown in Figure 2.

Overall, we have detected that over 22% of apps contain the "schedule" feature, while almost 17% of apps contain the "app management" feature. "Weather" is also a popular feature in about 12% of the apps, while "news" only resides in about 5% of all the apps. In total, we detected more than 3,200 features, averaging about 0.8 features for each app.

*3.4.1 Accuracy.* We evaluate the accuracy of the feature detection method through manually confirming each and every feature detected using our method. Because it is

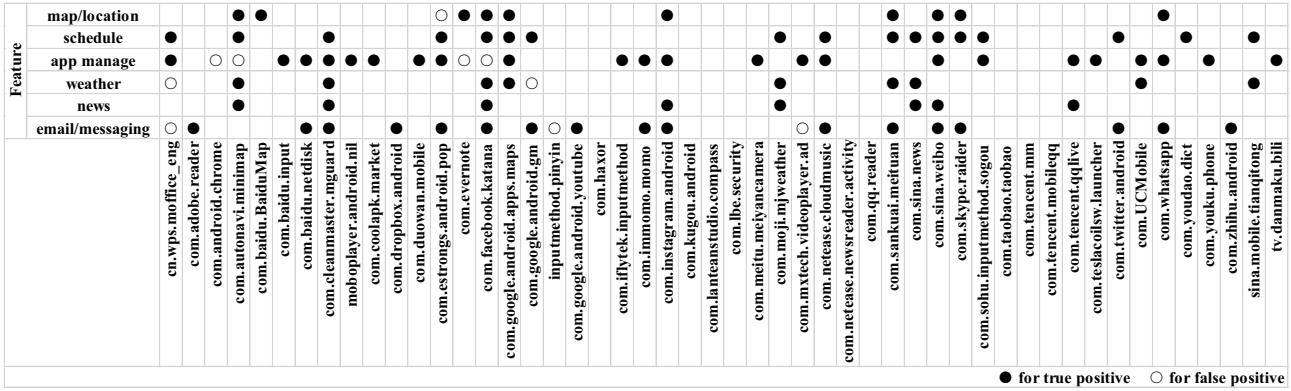



**Figure 3: Feature identification accuracy for 50 popular apps. All feature identified within each app is shown above and we also indicate each correct (●) and incorrect (○) features.**

**Table 3: Detection precision for each feature.**

| Feature | True Pos. | False Pos. | Precision |
|---|---|---|---|
| map/location | 10 | 1 | 90.9% |
| schedule | 17 | 0 | 100.0% |
| app manage | 23 | 4 | 85.2% |
| weather | 9 | 2 | 81.8% |
| news | 8 | 0 | 100.0% |
| email/messaging | 17 | 3 | 85.0% |
| **Average** | **84** | **10** | **89.4%** |

impossible to know the exact number of features within each app, we only evaluate the precision of our detection algorithm, leaving recall as future work.

We have evaluated the feature identification results for 50 popular apps based on the download statistics from CoolAPK manually. For each feature detected in an app, if there is any doubt whether the feature actually resides in the app, we check the data manually, then confirm whether it is a correct detection based on reading app description, running the app manually and checking the reasons why it is detected (whether the keywords are used for other purposes).

The evaluation results are shown in Figure 3. We show for each app all the features detected and whether it is correct or incorrect. We also summarize the overall detection accuracy data in Table 3. For all the 94 features identified from these 50 apps, 89% of them are correctly identified. Our detection algorithm is perfect on "schedule" and "news", while only have a 82% precision on detecting the "weather" feature.

Although the detection accuracy still has space to improve, we believe it is accurate enough, thus can be used to examine the extent of feature redundancy effectively.

*3.4.2 Number of Features in Each App.* When we look at individual apps, Facebook, Sina Weibo and CleanMaster have five features each, while many other apps include four features. Note that although we have identified all six features in Facebook, one of them (app management) is incorrect.

Figure 4 shows the distribution of the number of features detected in each app for all of 4,059 apps. We did not detect

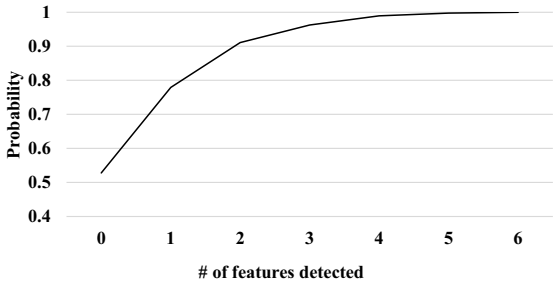



**Figure 4: Distribution of the number of features detected in each app.**

any of the six feature from just over 50% of the studied apps. Almost 22% of all the apps (900 apps) contain more than two features, while about 4% of the apps (150 apps) contain more than four features.

*3.4.3 Feature Overlapping.* We then examine how features overlap with each other in Table 4. For all apps detected with each feature (in each column), we show how many of other feature (in each row) can be detected in these apps.

Overall, the overlapping rate ranges from 10% to 49% for any pair of the six features. Take weather as an example, for all 475 apps containing the weather feature, 215 (45.3%) of them contain "schedule", while 140 (29.5%) of them contain "app management".

Some of the features overlap heavily with each other. For example, "schedule" has a high overlapping rate with "email/messaging", and vice versa. In contrast, "news" and "map/locatin" has a very low correlation, with their overlapping rate lower than 15% in both directions.

## 4 UNDERSTANDING FEATURE REDUNDANCY

With the list of features identified from over 4,000 popular apps, we then study how these features distribute on the apps installed on smartphones used by real users.

**Table 4: The number (and percentage) of apps overlapping with other apps. Each cell shows the number (and percentage) of each feature (in each column) in the apps containing another feature (in each row).**

| | map/location | schedule | app manage | weather | news | email/messaging |
|---|---|---|---|---|---|---|
| map/location | 249 (100%) | 109 (12.0%) | 93 (12.9%) | 84 (17.7%) | 29 (14.0%) | 84 (12.7%) |
| schedule | 109 (43.8%) | 911 (100%) | 245 (33.9%) | 215 (45.3%) | 97 (46.9%) | 321 (48.6%) |
| app manage | 93 (37.3%) | 245 (26.9%) | 723 (100%) | 140 (29.5%) | 61 (29.5%) | 237 (35.9%) |
| weather | 84 (33.7%) | 215 (23.6%) | 140 (19.4%) | 475 (100%) | 46 (22.2%) | 108 (16.3%) |
| news | 29 (11.6%) | 97 (10.6%) | 61 (8.4%) | 46 (9.7%) | 207 (100%) | 95 (14.4%) |
| email/messaging | 84 (33.7%) | 321 (35.2%) | 237 (32.8%) | 108 (22.7%) | 95 (45.9%) | 661 (100%) |

**Table 5: The different sets of users in our study.**

| Set | #Users | Types | Description |
|---|---|---|---|
| A | 600,000 | used | Users randomly taken from the stats of a popular app market |
| B | 60,000 | used | Top 10% of the users in Set A according to the number of apps used |
| C | 87 | installed | App lists collected from recruited volunteers |

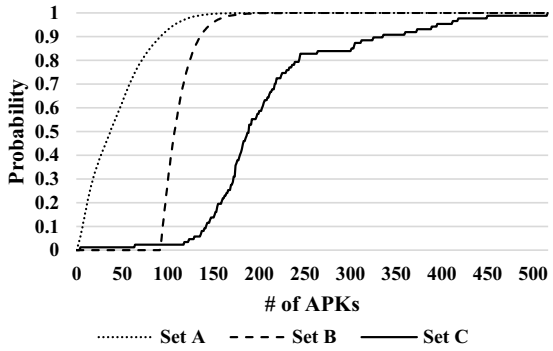



**Figure 5: The distribution of the number of apps in each dataset.**

## 4.1 Experimental Setup

*4.1.1 Datasets.* We use three sets of data in our study:

- **Set A.** One set of data is from a popular Android app market[4], which includes the list of apps used by each smartphone user during the period of one week. The data includes used app lists from 600,000 users.
- **Set B.** Heavy users from Set A. Because many users may not use many apps within one particular week, while users typically use fewer apps than the installed apps on their phones, we choose the top 10% of the users in the first dataset as the heavy users and study the extent of feature redundancy on these users. The dataset consists of 60,000 users and we expect that they can represent the typical heavy users with higher redundancy than Set A.
- **Set C.** Another set of data is collected with a simple Android app retrieving the list of apps installed on a smartphone, which was designed by ourselves. The users we recruited are mostly volunteers from universities. Most of them are college or graduate students, such that they know basic information on app features and are thus capable of answering a simple survey on whether these features are necessary for each app. We have recruited 87 volunteers who have returned valid app lists in this dataset.

Table 5 shows the details of the three sets of users. Note that there is one key difference between Set A/B and Set C: the first two datasets include only those apps used by each user during a certain period, which are fewer than the apps actually installed on the devices. Thus we use the third dataset to represent all the apps installed on each phone in the real world. Although the scale of the third dataset is pretty small, it will shed us insights on the scenario in reality.

*4.1.2 Dataset Comparison.* Figure 5 shows the difference in three datasets. In Set A, we can see that about half of the users have used fewer than 50 apps, while 90% of the users have used fewer than 90 apps during the period. In comparison, 90% of all users from Set C have installed more than 150 apps (many of them are pre-installed apps by the phone manufacturer), while 10% of the users have installed more than 350 apps!

Although the number in Set A is much lower than the statistics in Set C, the apps used by the users are typically popular apps. Many apps installed on smartphones may never be used, but they may contain redundant features and cause undesirable side-effects as well.

Set B was taken from the top 10% of the users in Set A, so all users in Set B have used at least 90 apps. Although we cannot retrieve a large number of installed app lists from real users, we expect that users in Set B can closely emulate a large set of installed apps from real users as in Set C.

## 4.2 Redundancy Calculation

In order to examine whether there are redundant features on a smartphone, we compare the list of apps on each smartphone with the list of 4,000 apps we have already studied in the last section. For each user, we record the features identified in all the apps matching the list of apps in our repository. We then count for each feature, how many of them can be found on every smartphone. If more than two apps contain

[4]http://www.wandoujia.com. All user identities are anonymized and appropriate measures are enforced to protect user privacy.

**Table 6: Distribution of the number of features on each smartphone.**

| Feature | Set A | Set B | Set C |
|---|---|---|---|
| map/location | 1.13 | 3.65 | 3.80 |
| schedule | 2.38 | 5.96 | 7.70 |
| app manage | 5.19 | 11.80 | 12.02 |
| weather | 2.07 | 4.63 | 4.82 |
| news | 1.48 | 3.63 | 2.40 |
| email/messaging | 2.75 | 7.05 | 8.47 |
| **Average** | **2.50** | **6.12** | **6.54** |

the same feature on one smartphone, then we have found a case of redundancy.

Because we do not rerun the feature identification algorithm for the dataset, we cannot identify features from apps other than the 4,000 apps we have already examined. As a result, the extent of redundancy in our study results might be slightly lower than the actual redundancy because we have not counted all the apps on a smartphone, instead only a subset of popular apps.

## 4.3 Results and Analysis

*4.3.1 Overall Results.* Table 6 shows for each feature, the average number of apps containing that feature found for all users in each dataset.

For all datasets, we can see that "app management" is the most redundant feature, averaging between 5 to 12 in the three datasets. The reason is because that Android apps installed from third-party markets or even the app's own websites typically check updates themselves. For one extreme case, we found over 100 apps containing app management features on a single smartphone.

"Weather" can be found on an average 2 to 5 apps on each smartphone, which is consistent with our intuition. "News" and "map/location" are the least popular among the six features across all the three datasets. However, these two features can also be sometimes very redundant on many smartphones, as we will show later.

When comparing the three datasets, because Set A includes only the list of apps used by the users during a certain period, instead of all the apps installed, the extent of feature redundancy is much lower compared to real world scenarios in Set C. However, when we choose the top 10% heavy users from Set A (to form Set B), the extent of redundancy is very close to the level of redundancy in Set C.

Table 7 shows the number of users with different numbers of redundant features from the three datasets. More than 85% of users from Set A contain at least one redundant feature on their phones, while all but one user (more than 98%) from Set C contain at least one redundant feature. Almost all users from Set B contain feature redundancy: it is not surprising because the users are top users selected from Set A.

More than 71% of all users from Set A contain multiple redundant features, while almost 18% of users contain all six redundant features. For Set C, 88% of all users contain more than five redundant features, while 61% of all users contain all six redundant features. When we consider only users with five or six redundant features, Set B has similar distribution to Set C, with 71% of user with six redundant features and 88% with five or more redundant features.

**Table 7: The number of users with different number of redundant features.**

| # of Redundant Features | Set A | Set B | Set C |
|---|---|---|---|
| 0 | 97,460 | 20 | 1 |
| 1 | 74,223 | 173 | 1 |
| 2 | 71,032 | 560 | 0 |
| 3 | 75,019 | 1,269 | 0 |
| 4 | 86,018 | 3,759 | 8 |
| 5 | 88,491 | 11,501 | 24 |
| 6 | 107,757 | 42,718 | 53 |
| **Total** | **600,000** | **60,000** | **87** |

*4.3.2 Feature Distribution.* Figure 6 shows the detailed distribution of each feature for the three datasets. All figures shown here are cumulative distribution functions (CDF) according to the number of features on each smartphone.

"App management" is obviously the most redundant feature, with more than 20% of the users from Set C having more than 16 apps with "app management" functions. "Schedule" and "email/messaging" are closely behind, with more than 20% of the users from Set C having more than 12 redundant apps with each feature. The other three features are less redundant, but still having 20% of the users with more than 5 apps for each feature.

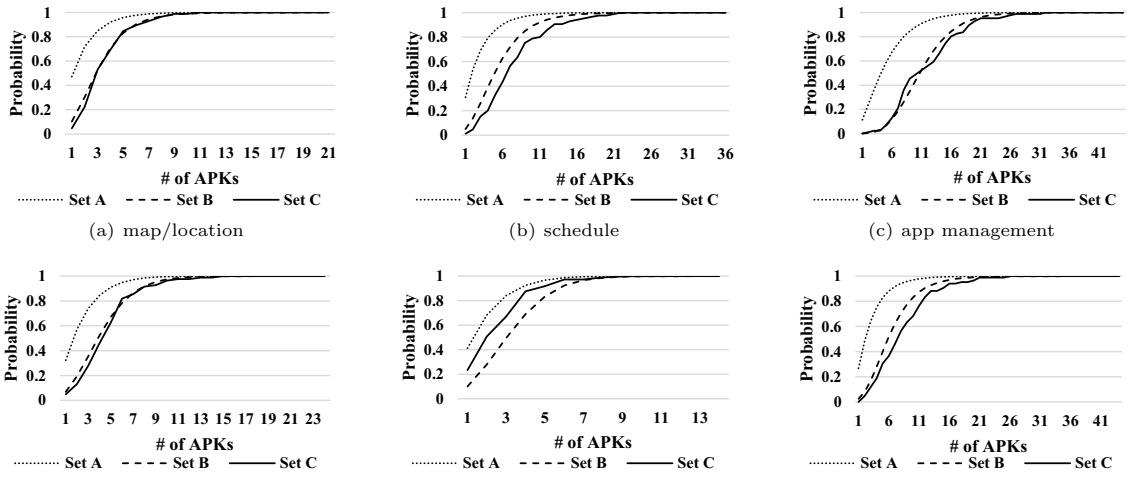
When we compare the difference between different sets, users from Set A obviously having the least redundancy for all six features. However, comparing Set B and Set C seems very interesting. Although Set B and Set C are very close in their redundancy distribution, there are some obvious differences in particular cases.

When we look at the features including "map/location", "app management" and "weather", their distributions are almost the same for Set B and C. For "schedule" and "email/message", the redundancy in Set C is obviously higher than in Set B, which indicates that many scheduling or messaging apps are not used very often. In contrast, the redundancy in "news" for Set B is higher than in Set C, which is counter-intuitive, but can be explained as well: (1) news apps installed on a smartphone are frequently used; or (2) fewer news apps are pre-installed by manufacturers.

# 5 USER SURVEY

After we identified the features on mobile apps, we then perform a simple user survey, asking each user to indicate which features they think are unnecessary for each app. We ask each user to provide their opinion on at most 10 apps.

Figure 7 shows two screenshots from the survey app. We first show the redundancy detection results, listing apps for each redundant feature. Then we ask the user to answer which features they deem as unnecessary, chosen from a list




(a) map/location (b) schedule (c) app management

(d) weather (e) news (f) email/messaging

**Figure 6: Distribution of redundancy for each feature in different datasets.**

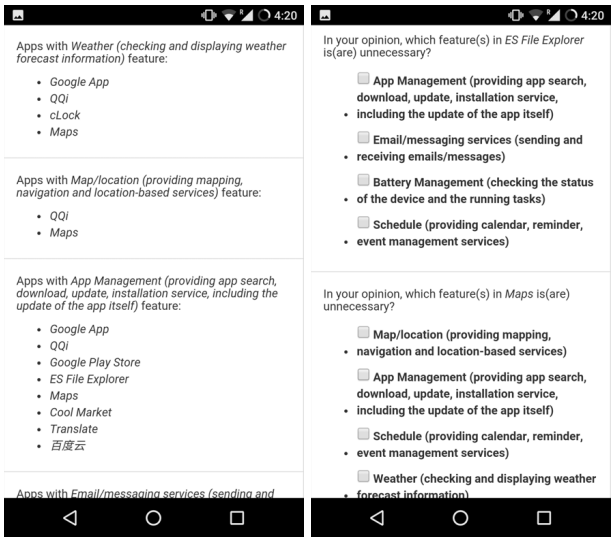



**Figure 7: Screenshots of the redundancy report result page and the survey questions.**

of features identified for popular apps. The features used in the survey are mostly top apps shown in Figure 3, where incorrect predictions are removed to improve the credibility of the survey.

The survey results are shown in Table 8, which shows the number of features listed in the questions and the number of features indicated as unnecessary. Overall, out of 1,336 features surveyed, users selected about 45% of them as unnecessary. For features such as "schedule", about 65% of the occurrences are deemed as unnecessary by users, while for "map/location", only 16% are considered as unnecessary.

The user responses confirms our speculation on the extent of redundancy existing on real smartphones. About half of the features we studied on smartphones are considered as redundant by users, especially those features involving scheduling, app management, email/messaging, and news services. Many users are surprised to find out that there are so much redundancy on their smartphones and suggest that these redundant features should be removed or consolidated.

**Table 8: Results from user survey, showing the number of features users selected as *unnecessary*.**

| Feature | # Asked | # Selected | % |
|---|---|---|---|
| map/location | 188 | 30 | 16.0% |
| schedule | 302 | 196 | 64.9% |
| app manage | 386 | 220 | 57.0% |
| weather | 109 | 38 | 34.9% |
| news | 116 | 59 | 50.9% |
| email/messaging | 235 | 122 | 51.9% |
| **Total** | **1,336** | **665** | **45.9%** |

# 6 DISCUSSIONS

## 6.1 Threats to Validity

**Feature definition.** As mentioned earlier, it is difficult to define what exactly is a feature. We did not attempt to give a definition, instead we compile a list of six common functionalities that we believe are important features, and studied them in popular Android apps. Some of these features are obvious, such as checking weather or app management. However, some of them are not so obvious. For example, "map/location" is probably too general as it covers many different functionalities in mobile apps. We realize this might be an issue and hope to clarify these in future study.

**Feature detection algorithms.** We understand there are limitations in the simple keyword-based method used in feature detection. However, we have tried many other feature detection methods that involve both static analysis and dynamic analysis, non of them have produced better results

than our current approach. As a matter of fact, we believe accurate feature location in software is still an unsolved issue, which requires more attention in our community.

**Dynamic features.** During feature identification, we can only detect features residing in UIs written in static code. However, many mobile apps nowadays are loaded dynamically or relying on WebView or HTML5. For example, we did not find any of our features from a popular messaging app WeChat, because most of its features are dynamically loaded during runtime. Many other apps load their main features from web servers. In order to detect features from these apps, we need to incorporate dynamic analysis techniques in the process, using tools such as Monkey or DroidBot [20].

### 6.2 False Redundancy

Even though we can identify that multiple occurrences of the same feature reside on one smartphone, it is sometimes difficult to tell whether these features are really "true" redundancy. Take weather as an instance, users might intentionally install multiple weather apps for special purposes. For example, people living in extreme climates or professional surfers or mountain climbers may in fact check many different weather apps on a regular basis.

This issue brings two implications: (1) Not all redundancies are bad or undesired redundancies. As indicated from our user survey, around 45% of the redundancies we have found are "true" redundancies from the users' perspective. (2) We need to find a way to distinguish between true redundancies and false redundancies, which needs further study.

### 6.3 Side-Effects of Feature Redundancy

Feature redundancy on a smartphone may bring many undesired side-effects, including occupying extra resources, wasting network traffic and consuming battery, etc.

Take weather forecast as an example, besides extreme cases, users usually only check weather information from the same app or widget, no matter how many apps check weather regularly. Thus checking weather forecast is a typical "true" redundancy, which not only wastes network data and power, but provides absolutely no valuable information.

Other redundant features such as app management and news services also involve wasted CPU and battery. In this paper, we did not calculate the wasted battery for each case, but it may be possible to do so once we understand the extent and frequency of the redundant operations.

### 6.4 Redundancy Elimination

As more and more apps are developed, we expect that the extent of feature redundancy is going to become worse in the future. Thus we need to consider remedy solutions to the feature redundancy problem.

Although it is recommended that users should remove those bloatware that are useless and merely wasting memory and other resources, many apps on smartphones are still useful apps, but they have been injected too many unwanted features. Simply uninstalling these apps is a straightforward, but no so effective, solution, as finding replacement for their main features are not always possible.

There are two types of redundancy: *unnecessary features* such as checking weather information, or *scattered features* such as app management and messaging services. Different redundant features may need different types of treatment. For example, unnecessary features such as weather information are mostly "true" redundancy and we typically only need to keep one weather forecast service on a smartphone.

Redundant features such as app management are different. We may need to manage and update each and every app on the phone, but we do not want each app to check for their own updates or even updates for all apps. One possible solution to this issue is *feature consolidation*, where we have a central service to control the update of all apps on a smartphone. Although Google recently requires all apps installed from Google Play can only update themselves through Google Play, it does not require apps installed from other parties to do the same. One we consolidate the feature in one single service, it will eliminate the extra checks, saving valuable network data and battery power as well.

For services such as email/messaging, it is a bit more complicated. We do not need multiple apps to check new emails from the same accounts, however messaging services are provided through different apps and their proprietary services. In the latter case, social network apps are one of the biggest power consumers on a smartphone. Even it is difficult to consolidate these services, it is still possible to apply power saving techniques such as network traffic delay [3, 5, 12], which can group all non-emergent network accesses to save network connection energy.

## 7 CONCLUDING REMARKS

This paper raises the issue of *feature redundancy* in mobile apps installed on a smartphone. With a feature identification method that is able to uncover common features from popular apps with a high accuracy, we studied the extent of feature redundancy in a large-scale dataset. Experimental results show that feature redundancy is prevalent on real smartphones, with more than 85% of them containing some kind of feature redundancy. Some common features such as "app management" are redundant by a dozen times on average. For the six common features we studied, all of them are redundant on at least half of the smartphones. User survey shows that about half of the redundant feature are indicated as unnecessary by smartphone users.

Our work represents the first step to understand feature redundancy on smartphones. We will investigate the issues and implications further in our future work and hope it will benefit both mobile users and app developers, while helping contribute towards a better mobile app ecosystem.

## ACKNOWLEDGMENT

This work was partly supported by the National Key Research and Development Program (2017YFB1001904) and the National Natural Science Foundation of China (61772042).

# REFERENCES


[1] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. 2009. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 280–293.

[2] Pierre F Baldi, Cristina V Lopes, Erik J Linstead, and Sushil K Bajracharya. 2008. A theory of aspects as latent topics. In *ACM Sigplan Notices*, Vol. 43. ACM, 543–562.

[3] Abhijnan Chakraborty, Vishnu Navda, Venkata N Padmanabhan, and Ramachandran Ramjee. 2013. Coordinating cellular background transfers using loadsense. In *Proceedings of the 19th annual international conference on Mobile computing & networking*. ACM, 63–74.

[4] Kunrong Chen and Václav Rajlich. 2000. Case study of feature location using dependence graph. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*. IEEE, 241–247.

[5] Xiaomeng Chen, Abhilash Jindal, Ning Ding, Yu Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone background activities in the wild: Origin, energy drain, and optimization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 40–52.

[6] Xin Chen and Sencun Zhu. 2015. DroidJust: automated functionality-aware privacy leakage analysis for Android applications. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 5.

[7] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Computer Security – ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–54.

[8] Android Developers. 2013. Intent. https://developer.android.com/reference/android/content/Intent.html. (2013).

[9] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.

[10] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. 2003. Locating features in source code. *IEEE Transactions on software engineering* 29, 3 (2003), 210–224.

[11] Whitson Gordon. 2014. Android Task Killers Explained: What They Do and Why You Shouldnt Use Them. http://lifehacker.com/5650894/android-task-killers-e xplained-what-they-do-and-why-you-shouldnt-use-them. (2014).

[12] Junxian Huang, Feng Qian, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. 2012. Screen-off traffic characterization and optimization in 3G/4G networks. In *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 357–364.

[13] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 1036–1046.

[14] Peng Huang, Tianyin Xu, Xinxin Jin, and Yuanyuan Zhou. 2016. DefDroid: Towards a More Defensive Mobile OS Against Disruptive App Behavior. In *MobiSys*. ACM, 221–234.

[15] Tetsuya Kanda, Yuki Manabe, Takashi Ishio, Makoto Matsushita, and Katsuro Inoue. 2013. Semi-automatically extracting features from source code of android applications. *IEICE TRANSACTIONS on Information and Systems* 96, 12 (2013), 2857–2859.

[16] Shinji Kawaguchi, Pankaj K Garg, Makoto Matsushita, and Katsuro Inoue. 2006. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software* 79, 7 (2006), 939–953.

[17] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49, 3 (2007), 230–243.

[18] Yuanchun Li, Yao Guo, and Xiangqun Chen. 2016. PERUIM: Understanding Mobile Application Privacy with permission-UI Mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. 682–693.

[19] Yuanchun Li, Yao Guo, Junjun Kong, and Xiangqun Chen. 2015. Fixing Sensor-Related Energy Bugs through Automated Sensing Policy Instrumentation. In *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED 2015)*. 321–326.

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. 23–26.

[21] Jialiu Lin, Shahriyar Amini, Jason I Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM, 501–510.

[22] Marcelo Martins, Justin Cappos, and Rodrigo Fonseca. 2015. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 563–575.

[23] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. 2008. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference*. ACM, 113–120.

[24] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 527–542.

[25] Julia Rubin, Michael I. Gordon, Nguyen Nguyen, and Martin Rinard. 2015. Covert Communication in Mobile Applications. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[26] Indrajeet Singh, Srikanth V Krishnamurthy, Harsha V Madhyastha, and Iulian Neamtiu. 2015. ZapDroid: managing infrequently used applications on smartphones. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 1185–1196.

[27] Steven Sinofsky. 2013. Avoiding mobile app bloat. https://blog.learningbyshipping.com/2013/09/24/avoiding-mobile-app-bloat/. (2013).

[28] Statista. 2018. Number of Apps in the Google Play Store. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. (2018). Accessed: 2018-02-06.

[29] TheNextWeb. 2014. Android users have an average of 95 apps installed on their phones, according to Yahoo Aviate data. http://thenextweb.com/apps/2014/08/26/android-users-average-95-apps-installed-phones-according-yahoo-aviate-data/. (2014).

[30] Kai Tian, Meghan Revelle, and Denys Poshyvanyk. 2009. Using latent dirichlet allocation for automatic categorization of software. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 163–166.

[31] Chengke Wang, Yao Guo, Yunnan Xu, Peng Shen, and Xiangqun Chen. 2016. Standby Energy Analysis and Optimization for Smartphones. In *IEEE MobileCloud 2016*.

[32] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-Phase Approach to Android App Clone Detection. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '15)*. ACM, 71–82.

[33] Wikipedia. 2018. Google Play. https://en.wikipedia.org/wiki/Google_Play. (2018). Accessed: 2018-02-06.

[34] Wikipedia. 2018. Software bloat. https://en.wikipedia.org/wiki/Software_bloat. (2018). Accessed: 2018-02-06.

[35] Norman Wilde and Michael C Scully. 1995. Software reconnaissance: Mapping program features to code. *Journal of Software: Evolution and Process* 7, 1 (1995), 49–62.

[36] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. 2013. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1043–1054.

[37] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY '12)*. 317–326.