Understanding the Inconsistencies between Text Descriptions and the Use of Privacy-sensitive Resources of Mobile Apps

Takuya Watanabe
Waseda University
3-4-1 Okubo Shinuku
Tokyo, Japan
watanabe@nsl.cs.waseda.ac.jp

Mitsuaki Akiyama
NTT Secure Platform Labs
3-9-11 Midoricho Musashino
Tokyo, Japan
akiyama.mitsuaki@lab.ntt.co.jp

Tetsuya Sakai
Waseda University
3-4-1 Okubo Shinuku
Tokyo, Japan
tetsuyasakai@acm.org

Hironori Washizaki
Waseda University
3-4-1 Okubo Shinuku
Tokyo, Japan
washizaki@waseda.jp

Tatsuya Mori
Waseda University
3-4-1 Okubo Shinuku
Tokyo, Japan
mori@nsl.cs.waseda.ac.jp

ABSTRACT

Permission warnings and privacy policy enforcement are widely used to inform mobile app users of privacy threats. These mechanisms disclose information about use of privacy-sensitive resources such as user location or contact list. However, it has been reported that very few users pay attention to these mechanisms during installation. Instead, a user may focus on a more user-friendly source of information: text description, which is written by a developer who has an incentive to attract user attention. When a user searches for an app in a marketplace, his/her query keywords are generally searched on text descriptions of mobile apps. Then, users review the search results, often by reading the text descriptions; i.e., text descriptions are associated with *user expectation*. Given these observations, this paper aims to address the following research question: *What are the primary reasons that text descriptions of mobile apps fail to refer to the use of privacy-sensitive resources*? To answer the research question, we performed empirical large-scale study using a huge volume of apps with our ACODE (Analyzing COde and DEscription) framework, which combines static code analysis and text analysis. We developed light-weight techniques so that we can handle hundred of thousands of distinct text descriptions. We note that our text analysis technique does *not* require manually labeled descriptions; hence, it enables us to conduct a large-scale measurement study without requiring expensive labeling tasks. Our analysis of 200,000 apps and multilingual text descriptions collected from official and third-party Android marketplaces revealed four primary factors that are associated with the inconsistencies between text descriptions and the use of privacy-sensitive resources: (1) existence of app building services/frameworks that tend to add API permissions/code unnecessarily, (2) existence of prolific developers who publish many applications that unnecessarily install permissions and code, (3) existence of secondary functions that tend to be unmentioned, and (4) existence of third-party libraries that access to the privacy-sensitive resources. We believe that these findings will be useful for improving users' awareness of privacy on mobile software distribution platforms.

Symposium on Usable Privacy and Security (SOUPS) 2015, July 22–24, 2015, Ottawa, Canada.

1. INTRODUCTION

Most applications for mobile devices are distributed through mobile software distribution platforms that are usually operated by the mobile operating system vendors, e.g., Google Play, Apple App Store, and Windows Phone Store. Third-party marketplaces also attract mobile device users, offering additional features such as localization. According to a recent report published by Gartner [1], the number of mobile app store downloads in 2014 are expected to exceed 138 billion. Mobile software distribution platforms are the biggest distributors of mobile apps and should play a key role in securing mobile users from threats, such as spyware, malware, and phishing scams.

As many previous studies have reported, privacy threats related to mobile apps are becoming increasingly serious, and need to be addressed [2, 3, 4, 5]. Some mobile apps, which are not necessarily malware, can gather privacy-sensitive information, such as contact list [6] or user location [7]. To protect users from such privacy threats, many of mobile app platforms offer mechanisms such as permission warnings and privacy policies. However, in practice, these information channels have not been fully effective in attracting user attention. For instance, Felt et al. revealed that only 17% of smartphone users paid attention to permissions during installation [4]. The Future of Privacy Forum revealed that only 48% of free apps and 32% of paid apps provide in-app access to a privacy policy [8]. Further more, Chin et al. reported that roughly 70-80% of end users ignored privacy policies during installation process [9].

Let us turn our attention to a promising way of communicating with users about apps and privacy. This information channel is the *text descriptions* provided for each app in a marketplace. The text description is usually written in natural, user-friendly language that is aimed to attract users' attention; it is more easily understood than the typical privacy policy. In addition, when a user searches for an app in a marketplace, s/he create query keywords, which are generally searched on text descriptions. Then, users review the

search results, often by reading the text descriptions; i.e., text descriptions can work as a proxy to the user expectations. In fact, text descriptions have a higher presence than permission warnings or privacy policies, and therefore, are a good channel for informing users about how individual apps gather and use privacy-sensitive information.

With these observations in mind, this work aims to address the following research question through the analysis of huge volume of Android applications:

> **RQ**: *What are the primary reasons that text descriptions of mobile apps fail to refer to the use of privacy-sensitive resources*?

The answers to the question will be useful for identifying sources of problems that need to be fixed. To address the research question, we developed a framework called ACODE (Analyzing COde and DEscription), which combines two technical approaches: static code analysis and text analysis. Using the ACODE framework, we aim to identifiy reasons for the absence of the text descriptions for a given privacy-sensitive permission. Unlike the previous studies, which also focused on analyzing the text descriptions of mobile apps [10, 11, 12, 13], our work aims to tackle with a huge volume of applications. To this end, we adopt light-weight approaches, static code analysis and keyword-based text analysis as described below.

Our static code analysis checks whether a given permission is declared. Then, it investigates whether the code includes APIs or content provider URIs [1] that require permission for accessing privacy-sensitive resources. Lastly, it traces function calls to check that the APIs and/or URIs are actually *callable* to distinguish them from apps with dead APIs/URIs that will never be used; e.g., reused code could include chunks of *unused* code, in which privacy-sensitive APIs were used.

Our description analysis leverages techniques developed in the fields of information retrieval (IR) and natural language processing (NLP) to automatically classify apps into two primary categories: apps with text descriptions that refer to privacy-sensitive resources, and apps without such descriptions. Here we present three noteworthy features of our approach. First, since we adopt a simple keyword-based approach, which is language-independent, we expect that it is straightforward to apply our text analysis method to other spoken languages. In fact, our evaluation through the multilingual datasets demonstrated that it worked for both languages, English and Chinese. Second, although our approach is simple, it achieves a high accuracy for nine distinct data sets. The accuracy is comparable to the existing pioneering work, WHYPER [11], which makes use of the state-of-the-art NLP techniques. The reason we developed the ACODE framework instead of using the WHYPER framework was that we intended to extend our analysis to multiple natural languages. The WHYPER framework leverages API documents to infer semantics. As of today, Android API documents are *not* provided in Chinese. Accordingly, we were not able to make use of the WHYPER framework to analyze Chinese text descriptions. Finally, like the WHYPER framework, our text analysis technique does *not* require manually labeled descriptions. Therefore, it enables us to enhance the text analysis of descriptions to any permission APIs without requiring expensive labeling tasks. It also enables us to reduce cost of text analysis significantly. The key idea behind our approach is to leverage the results of code analysis as a useful hint to classify text descriptions.

[1]Content providers manage access to data resource with permission using Uniform Source Identifiers (URIs); for instance, `android.provider.ContactsContract.Contacts.CONTENT_URI` is an URI used to get all users registered in the contact list.

To the best of our knowledge, only a few previous studies have focused on analyzing the text descriptions of mobile apps [10, 11, 12]. A detailed technical comparison between these studies and ours is given in section 7 (see Table 10 for a quick summary), and here we note that this work is distinguishable from other studies by being an extensive empirical study. The volume of our dataset is several orders of magnitude larger than previous studies. In addition, because we wanted to extract generic findings, we conducted our experiments in such a way as to incorporate differences in the resources accessed, market, and natural language. Our analysis considered access of 11 different resources taken from 4 categories, i.e., personal data, SMS, hardware resources, and system resources (see Table 1). We chose the resources because they are the most commonly abused or the potentially dangerous ones. We collected 100,000 apps from Google Play and a further 100,000 apps from third-party marketplaces. For the natural language analysis, we adopted English and Chinese, because they are the two most widely-spoken languages worldwide [14]. Furthermore, to evaluate the performance of text analysis, we obtained a total of 6,000 text descriptions from 12 participants. Each description was labeled by three distinct participants.

The key findings we derived through our extensive analysis are as follows:

The primary factors that are associated with the inconsistencies between text descriptions and use of privacy-sensitive resources are broadly classified into the following four categories.:

(1) **App building services/frameworks**: *Apps developed with cloud-based app building services or app building framework, which could unnecessarily install many permissions, are less likely to have descriptions that refer to the installed permissions.*

(2) **Prolific developers**: *There are a few prolific developers who publish a large number of applications that unnecessarily install permissions and code.*

(3) **Secondary functions**: *There are some specific secondary functions that require access to a permission, but tend to be unmentioned; e.g., 2D barcode reader (camera resource), game score sharing (contact list), and map apps that directly turns on GPS (write setting), etc.*

(4) **Third-party libraries**: *There are some third-party libraries that requires access to privacy-sensitive resources; e.g., task information (crash analysis) and location (ad-library, access analysis).*

The main contribution of our work is the derivation of these answers through the extensive analysis of huge volume of datasets. We believe that these findings will be useful for identifying sources of problems that need to be fixed to improve the users' awareness of privacy on mobile software distribution platforms. For instance, as our analysis revealed, there are several HTML5-based app-building framework services that unnecessarily install permissions, which could render the system vulnerable to additional threats of malicious JavaScript injection attacks. Therefore, an app developer should not install unnecessary permissions. However, if a developer used a rogue app-building framework service, he/she may likely not be aware of unnecessary permissions installed. ACODE enables operators of mobile software distribution platforms to pay attentions to these cases, which are invisible otherwise.

The rest of this paper is organized as follows. Section 2 describes our the ACODE framework in detail. In section 3, we show the details of the static code analyzer. Section 4 contains details of the text description classifier. We present our findings in section 5. Section 6 discusses the limitations of ACODE and future research directions. Section 7 summarizes the related work. We conclude

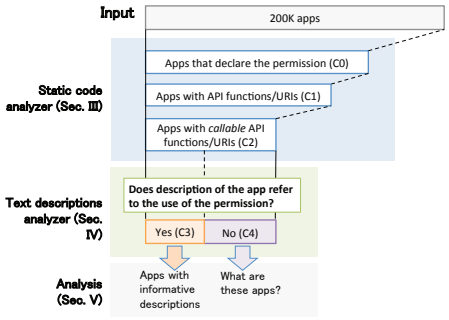

Figure 1: Overview of the ACODE framework.

Table 1: List of permissions used for this work.

Category	Permission	Definition*
Personal data	ACCESS_FINE_LOCATION	Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
	GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
	READ_CONTACTS	Allows an application to read the user's contacts data.
	READ_CALENDAR	Allows an application to read the user's calendar data.
SMS	READ_SMS	Allows an application to read SMS messages.
	SEND_SMS	Allows an application to send SMS messages.
Hardware resources	CAMERA	Required to be able to access the camera device.
	RECORD_AUDIO	Allows an application to record audio.
System resources	GET_TASKS	Allows access to the list of accounts in the Accounts Service . (This constant was deprecated in API level 21)
	KILL_BACKGROUND_PROCESSES	Allows an application to call killBackgroundProcesses(String).
	WRITE_SETTINGS	Allows an application to read or write the system settings.

*http://developer.android.com/reference/android/Manifest.permission.html

our work in section 8.

2. ACODE FRAMEWORK

In this section, we provide an overview of the ACODE framework. We also connect the components of the ACODE framework to the corresponding sections where we will give their details.

2.1 Goal and overview

Figure 1 is an overview of the ACODE framework. As discussed previously, we used a two-stage filter, employing a static code analyzer and text descriptions analyzer. In the first stage, the first filter extracted apps that declare at least one permission, e.g., location (C0). The second filter extracted apps with code that include corresponding APIs/URIs (C1). The third filter checked whether the APIs/URIs are *callable* from the apps by employing function call analysis (C2). In the second stage, the text classifier determined whether the text descriptions refer to the use of location explicitly or implicitly (C3), or not at all (C4). Note that we are not considering apps that do not declare to use permission, but have descriptions that indicate that permission is needed.

These filtration mechanisms enabled us to quantify the effectiveness of text descriptions as a potential source of information about the use of privacy-sensitive resources. For instance, by counting the fraction of apps that are classified as C3 (see figure 1), we can quantify the fractions of apps with text descriptions that successfully inform users about the use of privacy-sensitive resources for each resource. By examining the sources of apps that are classified as C4, we can answer our research question, **RQ**. The detailed analysis will be shown in section 5.

Figure 2 illustrates the components used in the ACODE framework. For each application, we had an application package file (APK) and a description. APK is a format used to install Android application software. It contains code, a manifest file, resources, assets, and certificates. The text descriptions of apps were collected from mobile software distribution platforms. As shown in the figure, the APKs and text descriptions were input to the static code analyzer and description classifier, respectively.

2.2 Static code analyzer

The goal of the static code analyzer is to extract APK files whose code include *callable* APIs/URIs that are required to use permissions related to a privacy-sensitive resource. For a given permission, first, we extracted apps that declare the use (C1, see section 3.1). Then, we checked whether disassembled code of the app include the APIs/URIs, which require the permission (C2, see section 3.2). If code included at least one API or URI, then, we checked whether it was actually *callable* within the app by investigating the function call graph with some heuristics we developed (C3, see section 3.3). It should be noted that the static code analysis has some limitations that we will discuss in section 6.

2.3 Description classifier

The goal of the description classifier was to classify text descriptions into two categories: those that refer to the use of a resource (C3), and those that do not (C4). In other words, we wanted to determine automatically whether a user can, by reading the text description, know that an app may use a privacy-sensitive resource. To do this, we leveraged several text analysis techniques. We also make use of the results of code analyzer to extract keywords associated with a resource. To extract keywords that are useful in classifying text descriptions, we first present text data preprocessing techniques in section 4.1. Next, in section 4.2, we present the keyword extraction method that leverages techniques used in the field of information retrieval. We also evaluate the accuracy of the description classifier in 4.3.

3. STATIC CODE ANALYSIS

This section describes the static code analysis techniques used in the ACODE framework. The purpose of static code analysis was to extract apps that include *callable* APIs/URIs to use a given permission. Before applying function call analysis, which is a process of checking whether given function is callable, we applied two filtration mechanisms: (1) permission filtration and (2) API/URI filtration. These filtrations are effective in reducing the computation overhead needed for function analysis. We also note that permission filter is useful to prune apps that include callable APIs/URIs, but will not actually use it.

3.1 Permission filtration

First, we applied permission filtration, which simply checks whether an app declares a given permission. According to Zhou et al. [15], permission filtration is quite effective in reducing the overhead of analyzing a huge amount of mobile apps. For each app, we investigated its AndroidManifest.xml file to check whether it declares permissions to access given resources. The process can be easily automated using existing tools such as `aapt` [16]. To further accelerate the data processing, we also leveraged multiprocessing techniques. Table 1 summarizes the 11 different permissions we analyzed in this work. To perform generic analysis, we chose the permissions from 4 categories, personal data, SMS, hardware resources, and system resources. These resources were chosen because they are the most commonly abused or the potentially dangerous ones.

3.2 API/URI filtration

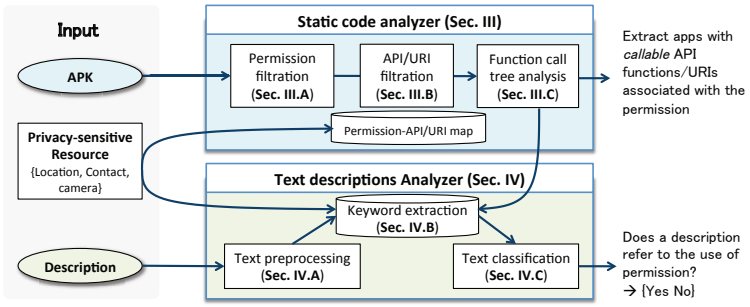

Figure 2: Components of the ACODE framework.

Next, for each sample, we checked whether it includes APIs or content provider URIs that require permissions to access privacy-sensitive resources. For this task, we made use of the API calls for permission mappings extracted by a tool called PScout [17], which was developed by Au et al. [18]. In addition to API-permission mapping, the PScout database also includes URI-permission mapping. To check the existence of APIs or URIs, first, using Android apktool [19], we extracted DEX code from APK files and disassembled them into smali format [20]. Then, we checked whether a set of APIs is included in the code of an APK file.

We note that some apps may require permissions but not include any APIs or URIs that request the permission. This may occur for several reasons. apps. If such possibly overprivileged apps are simply overprivileged due to developer's error, they do not impact our study, because those apps may not need to use APIs or URIs. However, as Felt et al. [3] reported, one of the common developer errors that cause overprivilege is Intent. A sender application can send an Intent to a receiver application, which uses permission API. In such cases, the sender of the Intent does not need to have permissions for the API. We saw many such cases, especially related to camera permissions. In fact, [3] reported that of the apps that unnecessarily request camera permission, 81% send an Intent to open the already installed camera applications (including the default camera) to take a picture. Our observation is in agreement with their finding.

Thus, our API/URI filtration scheme may miss a non-negligible number of apps that actually use the camera through Intent. However, note that our final analysis will be applied to the apps in set $C2$ as shown in figure 1. Therefore, we are confident that the removal of such apps should not affect our analysis, because we do not expect to see significant differences between the descriptions of those apps removed due to the Intent problem and the descriptions of apps included in $C2$.

3.3 Function call analysis

Now, we present the function call analysis of the ACODE framework. For convenience sake, let the term function include method, constructor execution, and field initialization; i.e., we trace not only method calls, but also class initializations. Figure 3 presents a pseudo-code of the algorithm we developed for function call analysis. It checks whether APIs/URIs of a given permission are callable (true) or not (false). The algorithm uses depth-first search to search the function call tree. If it finds a path from the given function to a class of `ORIGIN` (line 4), it concludes that the app has at least one API/URI that is callable, where `ORIGIN` is composed of three classes: `Application`, `App Components`, and `Layout`. `Application` is a class that initiates an Android app. It is called when an app is

```
1: INPUT
2: p : a permission
3: a : an application (APK)
4: ORIGIN = [Application, App Components, Layout]
5: list = getAU(p,a)  # list of APIs/URIs associated with p
6: done = []          # empty list
7:
8: WHILE list is not empty DO
9:     f = list.pop()
10:    IF f is in done:
11:        skip the function
12:    ENDIF
13:    IF f.parentClass is in ORIGIN:
14:        RETURN True
15:    ENDIF
16:    IF (f.parentClass inherits Android SDK)
17:        AND (f is not init)
18:        AND (f is not a static method):
19:        list.append(f.parentClass.init)
20:    ELSE IF (f is referenced):
21:        list.append(f.refFunctions)
22:    ENDIF
23:    done.append(f)
24: ENDWHILE
25: RETURN False
```

Figure 3: Pseudo-code that checks the callability of APIs of a permission.

launched. `App Components` are the essential building blocks that define the overall behavior of an Android app, including `Activities`, `Services`, `Content providers`, and `Broadcast receivers`. While the `Application` and `App Components` classes need to be specified in the manifest file of an app, the `Layout` class does not. It is often used by ad libraries to incorporate ads using XML.

`getAU` (Line 5) is a function that returns a list of APIs/URIs for a given permission. As an implementation of getAU, we adopted PScout [17]. `refFunctions` (line 21) is a function that returns a list of functions that reference to the given function or URI. As an implementation of `refFunctions`, we adopted androguard [21], which we modified to handle URIs. If a function of a class, say Foo, implements a function of the Android SDK class whose code is not included in the APK, we cannot trace the path from the function in some cases. To deal with such cases, we made a heuristic to trace the function that calls the init-method of class Foo (lines 16–19). We note that the heuristics can handle several cases such as async tasks, OS message handlers, or callbacks from framework APIs such as `onClick()`. A method is callable if it is overridden in a subclass or an implementation of the Android SDK and an instance of the class is created. Async tasks, the OS message handler, or

other callbacks implement their function by overriding the methods of the Android SDK subclass. Therefore, it should be handled by the heuristics.

4. TEXT DESCRIPTION ANALYSIS

This section describes the text description analysis used in the ACODE framework. The aim of this analysis was to classify descriptions into two classes: (1) text descriptions that reference a privacy-sensitive resource, and (2) text descriptions that do not. To this end, we adopted a set of basic techniques used in both IR and NLP fields. As we shall see shortly, our keyword-based approach is quite simple and works accurately for our task. As Pandita et al. [11] reported, a keyword-based approach could result in poor performance if it was designed naively. So, we carefully constructed our keyword extraction processes. As a result, we achieved 87-98% of accuracy for the combinations of 3 resources and two languages. Simple and successful text description classification enabled us to automate the analysis of 200,000 text descriptions.

Section 4.1 describes how we preprocessed the description data so that we can extract keywords that are useful in classifying text descriptions. Section 4.2 presents the keyword extraction method that leverages techniques used in the field of information retrieval. Section 4.3 describes our experiments to compare our description classifier with the WHYPER framework in terms of accuracy.

4.1 Text Data Preprocessing

To analyze natural language text descriptions, we applied several text preprocessing methods. These methods are broadly classified into four tasks; (1) generic text processing, (2) domain-specific stop words removal, (3) feature vector creation, and (4) deduplication. Especially the tasks (2) and (4) are crucial in extracting good keywords that can accurately classify the text descriptions.

4.1.1 Generic text preprocessing

We first apply widely-used generic text preprocessing techniques: word segmentation, stemming, and generic stop words removal. Word segmentation is a process of dividing text into words. This process is required for Chinese but not for English, in which words are already segmented with spaces. We used KyTea [22] for this task. For English, we applied stemming, which is a process of reducing derived words to their stem. It is known to improve the performance of text classification tasks. We used NLTK [23] for this task. Note that the concept of stemming is not applicable to Chinese. Lastly, we applied generic stop words removal, which is a process of removing a group of words that are thought to be useless for classification tasks because they are commonly used in any documentation (e.g., determiners and prepositions). As lists of stop words, we used the data in NLTK [23] for English and the data in imdict [24] for Chinese.

4.1.2 Domain-specific stop words removal

Next, we created domain-specific stop words list so that we can remove terms that are not generic stop words but are commonly used in mobile app descriptions; e.g., “app” or “free”. To this end, we make use of the technique proposed in Ref. [25], which is a term-based sampling approach based on the Kullback-Leibler divergence measure. Since the technique measures how informative a term is, we can remove the least weighted terms as the stop words. Number of sampling trial was set to 10,000. When we changed the threshold of extracting the top-L stop words; i.e., from $L = 20$ to $L = 150$, the following results are not affected at all. In the followings, we use $L = 100$. The extracted domain-specific stop words for English include “app”, “free”, “get”, “feature”, “android”, “like”, etc. Top-100 domain-specific stop words for English and Chinese are listed in Table 11 in the Appendix.

4.1.3 Feature vector creation

Using the preprocessed descriptions, we created a binary feature vector for each text description as follows. Let $\mathbf{W} = \{w_1, w_2, ..., w_m\}$ be a set of entire words after the screening process shown above. A feature of vector of the ith text description is denoted as $\mathbf{x}_i = \{x_i(w_1), x_i(w_2), ..., x_i(w_m)\}$, where $x_i(w_j) = 1$ if w_j is present in the ith text description. If w_j is not present, $x_i(w_j) = 0$.

4.1.4 Deduplication

Because we adopt the keyword extraction approach based on *relevance weights* as shown in the next subsection, the deduplication process plays a crucial role in eliminating the effect of same or similar descriptions generated by a single developer. For instance, if a developer produces thousands of apps with the same text description, which is often the case we observe in our datasets, the words included in the apps may cause unintended biases when computing the relevance weights of terms. To deduplicate the descriptions, we remove the same or similar descriptions by using the cosine similarity measure; i.e., for a given pair of feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, the cosine similarity is computed as $s = \cos\left(\mathbf{x}_i \cdot \mathbf{x}_j / |\mathbf{x}_i||\mathbf{x}_j|\right)$, and if s is larger than a threshold, the duplicated description is removed. We note that the value of threshold was not sensitive to the succeeding keyword extraction results if it is set between 0.5 to 0.8.

4.2 Keyword Extraction

To extract keywords, we leverage the idea of *relevance weights*, which measures the relation between the relevant and non-relevant document distributions for a term modulated by its frequency [26]. Relevance weighting was developed in the IR community as a means to produce optimal information retrieval queries. To make use of the relevance weights for our problem, we need to have sets of relevant and non-relevant documents. Since we do not have any labels that indicate whether a document is relevant, i.e., it refers to a permission, or non-relevant, i.e., it does not refer to a permission, we set the following assumption.

Assumption: *For a given permission, descriptions of apps that declare the permission and have callable APIs can be regarded as “pseudo relevant document”, while the descriptions of the remaining apps can be regarded as “pseudo non-relevant document”.*

Note that our research question contradicts with this assumption; i.e., we are interested in the reason why an app with callable API for a permission does not refer to the permission. Nevertheless, our performance analysis using multiple permissions in two spoken languages empirically supports that our approach actually works well in extracting effective keywords.

Under this assumption, we calculate the relevance weights for each word as follows. For a word w_i, the relevance weight (RW) is

$$RW(w_i) = \log \frac{(r_i + 0.5)(N - n_i - R + r_i + 0.5)}{(n_i - r_i + 0.5)(R - r_i + 0.5)},$$

where r_i is the number of relevant documents word w_i occurs in, R is the number of relevant documents, n_i is the number of documents word w_i occurs in, and N is the number of documents, respectively.

Using the entire descriptions with code analysis outputs, we extracted the keywords that have the largest relevance weights. Table 2 presents a subset of extracted keywords for each permission. For space limitation, we present only the Top-3 English keywords. We have listed the top-10 keywords for English and Chinese in Table 12 in the Appendix. In most cases, the keywords look intuitively reasonable. Interestingly, some keywords such as “sms”

Table 2: Extracted top-3 keywords for English descriptions.

Resources	1st	2nd	3rd
Location	gps	location	map
Account	grab	google	youtube
Contact	sms	call	contact
Calendar	calendar	reminder	meeting
SMS (read)	sms	message	incoming
SMS (send)	sms	message	sent
Camera	camera	scan	photo
Audio	recording	voice	record
Get tasks	lock	security	task
Kill background process	task	kill	manager
Write setting	alarm	ring	bluetooth

are found in multiple resources; i.e., contact, SMS (read), and SMS (send). In fact, these resources tend to co-occur. In the following, we will use these keywords to classify descriptions. Once we compiled the keywords, the text classification task is straightforward. If a text description includes one of the extracted keywords for a permission, the description is classified as positive, i.e., it refers to the permission. The problem is how we set the number of keywords to be used. We will study the sensitivity of the threshold in Section 4.3.2.

4.3 Performance Evaluation

To evaluate the accuracy of our scheme, we use manually labeled data sets. We first present the way how we compile the labeled data set. Next, we evaluate the accuracy of our approach, using the labeled data. Finally, to validate the robustness of our approach, we use the external dataset and compare the performance with the existing state-of-the-art solution, the WHYPER framework. In the analysis of accuracy (Section 4.3.3), we use 200,000 apps, which will be described in Section 5.1 as *training sets*; i.e., they are only used for keyword extraction. The *labeled test set* is a subset of those, on which we measure accuracy. We note that in the evaluation, our training set included test set; i.e., we extracted the keywords using the entire text descriptions, which is the training set, and applied the keywords (i.e., classifier) to the labeled descriptions, which is the test set. In general, training classifier using test set is not good because such setting could over-estimate the accuracy of the model. However, the effect should be small because our classifier was based on frequencies of terms and the test set accounted for only 0.6% of entire samples.

4.3.1 Creation of labeled datasets

We created the labeled data sets with the aid of 12 international participants who are from China, Korea, Thailand, and Indonesia. All the participants were university students with different disciplines in science and engineering. 7 were female and 5 were male. 4 were native English speakers, and 8 were native Chinese speakers. None of them had experience of developing Android applications. All the native Chinese speakers were fluent in English (native level). Students who were native speakers of Chinese labeled Chinese descriptions. In summary, six students labeled English descriptions, and the other six labeled Chinese descriptions. Here, we picked up three distinct resources, i.e., location, contact, and camera, out of the 11 resources we considered in this work.

Since a resource is used for various purposes, and referred to by various terms, we wanted to avoid participants focusing too much on a particular keyword, such as "camera". Instead, we asked participants to identify whether an app will use a camera, rather than whether it mentions a camera. This enabled us to identify several interesting keywords, such as "QR" and "scan". Also, we note that the question should reflect users' *awareness* of a resource.

Table 3: Summary of labeled datasets.

English			
	Location	Contact	Camera
# of descriptions	1,000	1,000	1,000
# of labels	3,000	3,000	3,000
Chinese			
	Location	Contact	Camera
# of descriptions	1,000	1,000	1,000
# of labels	3,000	3,000	3,000

Table 4: Statistics of labeled descriptions to be used for performance evaluation.

English			
	Location	Contact	Camera
# of positive descriptions	128	208	276
# of negative descriptions	611	449	289
Chinese			
	Location	Contact	Camera
# of positive descriptions	38	102	157
# of negative descriptions	828	544	583

Before asking participants to label text descriptions, we picked some descriptions from our entire data set. If random sampling were applied to the entire set, there would be a significant imbalance between the two classes. In particular, there would be very few positive samples, i.e., text descriptions that reference a resource. To avoid such an imbalance, we applied the access permission filter shown in section 3.1 so that the sampled text descriptions would include a certain number of positive samples. Although this solution could create some bias toward the positive class, in fact it did not matter, as will be shown later in this paper. From the set of apps that declare access permissions for using resources, we randomly sampled 1,000 text descriptions. In total, we sampled 6,000 descriptions, as shown in table 3.

Having sampled text descriptions, we asked each participant to label 500 text descriptions for each resource (e.g., $500 \times 3 = 1,500$ descriptions in total). A participant labeled text descriptions in either English or Chinese. To increase the quality of labels, each text description was labeled by three distinct, fixed participants. We obtained a total of $18,000$ labels for $6,000$ text descriptions, as shown in table 3.

Finally, we eliminate inconsistent labels to ensure that the quality of labels is high; i.e., we used only the text descriptions upon which all three evaluators agreed. Table 4 summarizes the text descriptions that met this criterion. We used these labeled descriptions for evaluating accuracy of our approach, as described in the next subsection.

4.3.2 Threshold Sensitivity Study

Using the labeled datasets, we empirically studied the relation between threshold and classification accuracy. Here, the definition of the accuracy is the fraction of correctly classified text descriptions, using the top-K keywords. Figure 4 presents how the number of keywords, K is correlated with the classification accuracy. As shown in the graph, across the 6 of labeled datasets, the accuracy is fairly stable around $K = 3$. Also, we notice that $K = 3$ gives the highest accuracy with the minimum variance. As we increase K, the accuracy is degraded; i.e., as K increases, the less relevant the keywords become. In fact, while many of keywords listed in

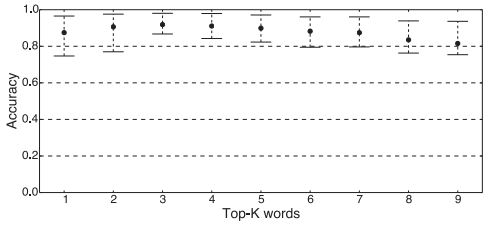

Figure 4: K vs. accuracy. The circles indicate median values and the bars indicate maximum/minimum values, respectively.

Table 5: Accuracy of our approach ($K = 3$) for the 6 of labeled datasets.

Resource	Lang	TP	TN	FP	FN	ACC	PPV	NPV
Location	EN	118	591	20	10	0.959	0.855	0.983
	CN	23	826	2	15	0.980	0.920	0.982
Contact	EN	177	396	53	31	0.872	0.770	0.927
	CN	64	535	9	38	0.927	0.877	0.934
Camera	EN	206	284	5	74	0.867	0.976	0.802
	CN	98	575	8	59	0.909	0.925	0.907

Table 12 in the Appendix look natural, some lower-ranked keywords such as "gps" for `SEND_SMS` or "call" for `READ_CALENDAR` do not really make sense. Given these observations, in the following analysis, we adopt $K = 3$ in classifying the document. We note that the chosen threshold works nicely for the external dataset provided by the authors of WHYPER [11]. We will report the results in Section 4.3.4.

4.3.3 Accuracy of Text Classification

We now evaluate the accuracy of our text classifier. To measure the accuracy, we use several metrics. First, TP, TN, FP, and FN represents number of true positives, number of true negatives, number of false positives, and number of false negatives, respectively. We also use three derivative metrics: accuracy (ACC), Positive predictive values (PPV), and Negative predictive values (NPV), which are defined as

$$ACC = \frac{TP+TN}{TP+TN+FP+FN},$$
$$PPV = \frac{TP}{TP+FP},\ NPV = \frac{TN}{TN+FN},$$

respectively. PPV and NPV measure how many of descriptions classified as positive/negative are actually positive/negative. These measures are suitable to our requirements because we aim to derive the answers of our research question by studying the characteristics of classified descriptions. Therefore, we expect that these measures have high values.

Table 5 presents the results of performance evaluation. In both languages, the observed accuracy was good for all categories; e.g., ACCs were 0.87–0.98. Also, in most cases, NPVs were larger than 0.9. Since one of our objectives is to understand the reasons why text descriptions fail to refer to access permissions, the high number of NPVs is helpful, because it indicates that majority of descriptions classified as negative are actually negative. In summary, our scheme was validated to enable automatic classification of text descriptions into the two categories with good accuracy. It works well for both languages, English and Chinese.

4.3.4 Robustness

Table 6: Statistics of the WHYPER datasets.

	Contact	Calendar	Audio
# of positive samples	107	86	119
# of negative samples	83	110	81

Table 7: Comparison of accuracy of ACODE ($K = 3$), WHYPER semantic analysis (WHYPER), and WHYPER keyword (WKW).

Resource	method	TP	TN	FP	FN	ACC	PPV	NPV
Contact	ACODE	96	63	20	11	0.837	0.828	0.851
	WHYPER	92	77	6	15	0.889	0.939	0.837
	WKW	95	46	37	12	0.742	0.720	0.793
Calendar	ACODE	77	98	12	9	0.893	0.865	0.916
	WHYPER	81	99	11	5	0.918	0.880	0.952
	WKW	84	60	50	2	0.735	0.627	0.968
Audio	ACODE	95	57	24	20	0.742	0.720	0.793
	WHYPER	103	69	12	16	0.860	0.896	0.812
	WKW	113	38	43	6	0.755	0.724	0.864

To validate the robustness of our approach, we use the external labeled dataset [27], which is provided by the authors of the WHYPER framework [11]. Since the dataset also includes the outcomes of the WHYPER framework, we can directly compare the performance of the two frameworks. Since the dataset consists of a set of labels for each sentence, we reconstructed original descriptions from the sentenses and assign labels to the descriptions; i.e., if a description consists of at least one sentence that declares the use of a permission, the description is labeled as positive, otherwise labeled as negative. Table 6 summarizes the dataset[2]. All the descriptions are written in English.

Table 7 shows the comparison of performance of the ACODE framework and the WHYPER framework in classifying descriptions. Our results show that the performance of the ACODE framework is comparable with that of the WHYPER framework. Especially, the delta for NPV, which is the most important metrics for our study, is less than 0.04 for all the three cases. We also notice that the keyword-based approach used in the WHYPER paper (WKW in the table) had high false positives. We conjecture that the high false positives are due to the nature of extracted keywords, which include some generic terms such as data, event, and capture.

Notice that the WHYPER dataset consists of higher fractions of positive descriptions, compared to ours. This may reflect the fact that the apps used for WHYPER study were collected from the top-500 free apps; i.e., it is likely the top apps were built by skilled developer and had informative descriptions. In contrast, our datasets consist of larger fractions of negative samples. Since our datasets were collected from entire app space, they consist of various apps, including the ones that failed to add informative descriptions due to the reasons that will be described in the next section. Despite this potential difference in the population of datasets, our framework established good accuracy among all the datasets.

In summary, we evaluated the accuracy of the ACODE framework using 5 of 11 permissions we considered[3]. In the following large-scale analysis, we assume that the ACODE framework establishes good accuracy for the rest of permissions as well. The

[2]We derived these numbers by analyzing the dataset [27]

[3]To be precise, we verified 5 of 11 permissions for English and 3 of 11 permissions for Chinese.

Table 8: Summary of Android apps used for this work.

	English	Chinese	Data collection periods
Official (Google Play)	100,000	0	Apr 2012 – Apr 2014
Third-party (Anzhi)	0	74,506	Nov 2013 – Apr 2014
Third-party (Nduoa)	0	25,494	Jul 2012 – Apr 2014

potential effect of the assumption will be discussed in Section 5.5.

5. ANALYSIS OF CODES AND DESCRIPTIONS

Using the ACODE framework, we aim to answer our research question **RQ** shown in Section 1. We first describe the details of the data sets we used for our analysis, in section 5.1. Then, we apply our code analysis to the apps and extract apps with *callable* APIs/URIs of permissions ($C2$, see figure 1) in section 5.2. Using the extracted apps with callable APIs/URIs of permissions, section 5.3 aims to quantify the fractions of apps with text descriptions that successfully inform users about the use of privacy-sensitive resources for each resource. In section 5.4 we aim to answer the research question **RQ**. We discuss in-depth analysis to understand the reasons of failures for text descriptions classified as $C4$ in informing users about access permissions. Finally, Section 5.5 discusses the limitations of our analysis and evaluation.

5.1 Data sets

We collected Android apps from the official marketplace [28] and two other third-party marketplaces [29, 30]. All these marketplaces have huge user bases. Note that these were all free apps. Although we might see some disparity between free and paid apps, we leave this issue open for future research.

After collecting mobile apps, we first pruned samples that are corrupt or have zero length text descriptions. From the rest of the samples, we randomly picked 100,000 apps for each type of markets. Table 8 summarizes the data sets we collected. Among 200,000 apps, only 1,831 apps were duplicated in package names between the two markets. To simplify the interpretation of analyses, we assigned different languages, English and Chinese, to the official and third-party marketplaces. Note that we have already shown that our text description classification scheme works well for both languages.

5.2 Extracting apps with callable APIs/URIs of privacy-sensitive resources

Table 9 presents the results of our code analysis. Overall, many applications require permission of location. As we will detail later, many of these are apps that use ad libraries. Interestingly, the popularity of personal data resource requirements is almost identical across markets. The most popular is location, second is contact, third is accounts, and fourth is calendar. Generally, third-party markets tend to require/use more permissions than the official market. This may correlate to the existence of defense mechanisms installed on the official marketplace – Bouncer [31].

Another useful finding we can extract from the results is that over privilege ($C0 - C1$) is observed commonly across the categories. Also, there are non-negligible numbers of apps that have code to use permissions but cannot be called ($C1 - C2$). This often occurs when a developer incorporates an external library into an app; the library has many functions, including APIs/URIs of permissions, but the app does not actually call the APIs/URIs. Our code analysis can prune these applications from further analysis.

Overprivilege ratios are especially high for account and contact permissions in the third party marketplaces and for camera, calendar, and kill background processes permissions in both markets. Careful manual inspection revealed that these cases can be attributed to misconfiguration on the part of developers; i.e., the Intent issue discussed in section 3.2. Such apps were pruned by the second filter. We also note that these apps do not need to declare permissions because the permissions are misconfigurations. These observations agree with the work performed by Felt et al. [3]. Although our scheme pruned those applications, the pruning did not affect the analysis because the pruned apps are unlikely to exhibit special characteristics in their text descriptions.

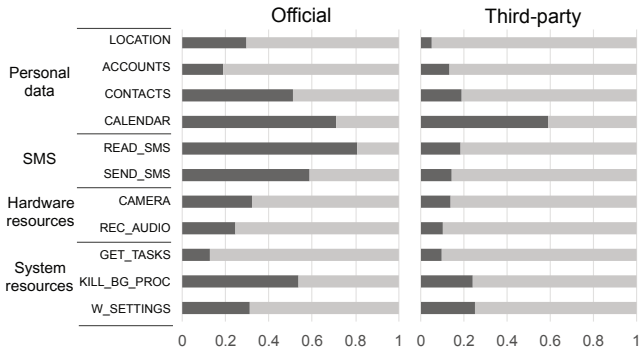

Figure 5: Fractions of descriptions that refer to a permission. Populations are $C2$ apps shown in Table 9; e.g., of the 18,165 of official market apps with callable functions that request location permission, roughly 30% of them mentioned the use of location in the description.

5.3 Analysis of apps with callable APIs/URIs for a permission.

Using apps that include callable APIs/URIs for a permission ($C2$ in Table 9), we analyzed their text descriptions. Figure 5 presents the results. We first notice that fractions of positive text descriptions are higher for official market apps. This can be considered natural, given that official market is more restrictive. We also notice that some resources such as `CALENDAR` for both markets and `SMS` permissions and the `KILL_BG_PROC` (kill background process) permission for the official market are well described in their descriptions.

For the official market, `GET_TASK` and `ACCOUNTS` were the permissions that were less described (15–20%). In contrast, `READ_SMS` and `CALENDAR` were the permissions that were well described (70–80%). These results are consistent with intuition that permissions that are directly associated with user actions tend to be well described. Overall, our impression is that for the official market, the fractions of proper descriptions are higher than expected. Thus, if the descriptions of remaining apps were improved, the text description could serve as a good source of information to let users know about sensitive resources.

Finally, we note that the descriptions of apps collected from official market was only English, while the descriptions of apps collected from third-party market was only Chinese. Therefore, we cannot tell if the observed differences are due to the market or the language. We leave the issue for future work.

5.4 Answers to the Research Question

To answer the research question **RQ**, we performed the manual inspection to the extracted apps that fail to refer to use of permis-

Table 9: Numbers of extracted apps for each category.

Official market apps											
	Personal data				SMS		Hardware resources		System resources		
	Location	Accounts	Contacts	Calendar	SMS (read)	SMS (send)	Camera	Audio	Get tasks	Kill bg processes	Write setting
Permission (C0)	25026	9943	6962	1893	1352	3471	10232	5204	4646	409	2873
API/URI (C1)	23390	6948	6177	333	526	2043	7173	4621	3433	248	1954
Callable (C2)	18165	3933	4238	100	287	1567	6141	3297	1737	208	1744
Third-party market apps											
	Personal data				SMS		Hardware resources		System resources		
	Location	Accounts	Contacts	Calendar	SMS (read)	SMS (send)	Camera	Audio	Get tasks	Kill bg processes	Write setting
Permission (C0)	40278	6585	9907	394	7686	16204	14581	10745	37436	7457	15249
API/URI (C1)	36885	3148	4863	98	4668	13807	6934	8354	19147	1158	11564
Callable (C2)	32122	1542	3429	66	4185	12355	6139	6147	15447	957	1029

sions. The methodologies of the manual inspection are described below. Given a permission, e.g., `Camera`, we fist identify Java classes that include the APIs associated with the permission. From the identified class, we can extract a package name such as `/com/google/android/foo/SampleCameraClass.java`, which is segmented into a set of words, com, google, android, foo, and SampleClass. By analyzing the package name words for apps that fail to refer to use of the permission, we can find intrinsic words that are associated with specific libraries such as "zxing" used for handling QR code or service names such as "cordova", which is an app building framework. In addition, we can analyze developer certificates included in app packages. We also apply dynamic analysis of the apps when we need to check how the permission is used. Using the methodologies, we classified such apps into the four categories. For each category, we extracted reasons why text descriptions fail to refer to permissions.

(1) **App building services/frameworks**

Through the analysis of package names of apps, we noticed that many of apps were developed with cloud-based app building services, which enable a developer to create a multi-platform app without writing code for it. Examples of cloud-based app building services are SeattleClouds, iBuildapp, Appsbar, appbuilder, and biznessapps. Similarly, many of apps were developed with mobile app building frameworks, which also enable a developer to create a multi-platform app easily. Examples of such mobile app building frameworks are Apache Cordova (Phonegap) and Sencha. These services/frameworks provide a simple and intuitive interface to ease the processes of building a mobile app.

Among many such services/frameworks, we found a few services/frameworks that generate apps that *unnecessarily* install many permissions, and put callable APIs/URIs for the permissions into the code. Since a developer using such a service/framework cannot change that setting, it is likely that even the developer is not aware of the fact that app install the permissions with callable APIs/URIs; hence, it is less likely the developer writes about the permissions in the description.

Figure 6 shows CDFs of number of permissions per application. First, apps collected from the official market have small number of permissions among the 11 permissions; i.e., more than 80% of apps had zero permissions. They had other generic permission such as Internet. Second, we considered an intrusive cloud-based app building service and one of the popular app building frameworks. Both cases tend to install a large number of permissions. Especially, roughly half of the apps that were built with the intrusive cloud-based app building service had a fixed number of permissions (4 out of 11). We carefully inspected these apps, and found that many permissions such as record audio were unnecessarily installed by the services/frameworks.

We revealed that the apps built by the intrusive cloud-based app building services are popular in official market, but not popular in third-party market. In the official market, more than 65% of apps

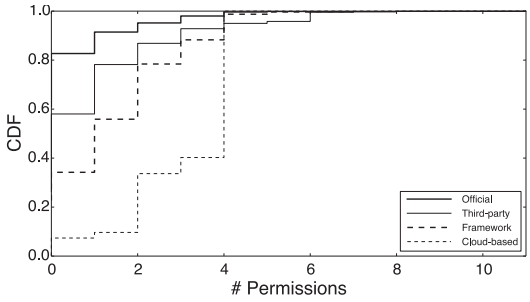

Figure 6: CDFs of number of permissions per application. The 11 permissions listed in Table 1 are used.

that failed to refer to use of `record audio` were developed with these services. Similarly, more than 25% of apps that failed to refer to use of `contact list` were developed with these services. We also observed non-negligible number of such apps in other resources; i.e., 5% for location and 10% of camera. For app building frameworks, one of the frameworks accounted for more than 28% of apps that failed to refer to use of `record audio` in the third-party market. In fact the permission was not necessary for the apps.

We also note that unnecessarily installed permissions on a framework such as phonegap, which is HTML5-based mobile app building framework, could bring additional threats because such permission can be abused through various channels of Cross-Site Scripting attacks [32].

(2) **Prolific developers**

Through the analysis of distributions of number of apps per developer certificate, we noticed that a very few number of developers accounted for a large number of descriptions without mention of privacy-sensitive resources. We call such developers "prolific developers". For instance, five prolific developers published 47% of third-party market apps that fail to refer to `send SMS`. We applied eleven popular commercial anti-virus scanners to the apps with SMS permission, and checked whether either of scanner detected the types of application. If at least one scanner detected an app as malware/adware, we marked it as malware/adware. We found that majority of the apps with unmentioned SMS permission were malware/adware and have been removed from the market later. There are other cases. Three prolific developers published 38% of third-party market apps that fail to use of `kill background processes`. Another three prolific developers published 32% of third-party market apps that fail to use of `write setting`. We carefully inspected these apps, and found that they do not have any reasons to use the permissions. Although not conclusive, we conjecture that these prolific developers likely reuse their own code for building a large number of apps; i.e., they tend to include unnecessary permissions/code.

(3) **Secondary functions**

Through the careful analysis of descriptions that failed to refer to permissions, we found several secondary functions that tend to be unmentioned. For instance, several apps have functions to share information with friends, e.g., scores of games. In many cases, such functions require to access contact list. However, such activity is often unmentioned in the descriptions because it is an optional function. Another example is map-based apps that require to access the `write setting` permission to enable location positioning service such as GPS or Wi-Fi. Such map-based apps accounted for 44% of apps that failed to refer to `write setting`. Among several cases, the most notable one was barcode reader, which requires access to `camera` device. Although there are several barcode reader apps, majority of apps with barcode reader function are shopping apps or social networking apps. Since the barcode reader is not a primary function for those apps, it tends to be unmentioned in their descriptions. To study the impact of such cases, we extracted apps that use barcode libraries such as ZXing [33] or ZBar [34]. We found that in the official market, more than 53% of apps that failed to refer to use of camera had barcode reader libraries in their code. In the third-party market, more than 66% of such apps had barcode libraries. Mobile application distribution platform providers may want to support exposing the use of privacy-sensitive resources by functions that tend to be unmentioned.

(4) **Third-party libraries**

There are some third-party libraries that need to use privacy-sensitive resources. For instance, it is well known that ad libraries make use of resources of location or account information for establishing targeted advertisement [5]. Another example of third-party libraries are log analysis libraries and crash analysis libraries. These libraries make use of `get task` permission and location information. We analyzed apps that have callable location APIs/URIs and text descriptions that do not refer to the location permission. We found that in the official market, more than 62% of such apps use ad libraries. In the third-party market, more than 80% of such apps used ad libraries. Similarly, in the third-party market, more than 20% of apps that failed to refer to location permission used access analysis libraries. Thus, if a developer uses these third-party libraries, it is likely that the description of the app fails to refer to the permission unless the developer explicitly expresses it.

5.5 Threats to Validity

This section discusses several limitations of our analysis and evaluation.

5.5.1 Static code analysis

Although we developed an algorithm to check whether privacy-sensitive APIs/URIs are callable, we are aware of some limitations. First, although the algorithm can detect the callability of APIs/URIs, we cannot precisely ensure that they are actually called. Second, our static code analysis cannot dynamically track assigned program code at run-time, such as reflection. Third, as Poeplau et al. [35] revealed, some malware families have the ability to self-update; i.e., after installation, an app can download the new version of itself and load the new version via `DexClassLoad`. Employing dynamic code analysis could be a promising solution to these problems. However, other challenges may include scalability and the creation of test patterns for UI navigations [36, 37]. As we mentioned earlier, we adopted static analysis because our empirical study required analysis of a huge volume of applications. On the other hand, we note that static code analysis has a chance to extract *hidden* functions that cannot be explored by a dynamic analysis. We leave these challenges for our future work.

5.5.2 Accuracy of the keyword-based approach

As we mentioned earlier, we evaluated the accuracy of the ACODE framework using 5 of 11 permissions we considered. Our assumption is that the ACODE framework establishes good accuracy for the rest of 6 permissions. However, there may be a concern that the keyword-based approach works better for some permissions more than others. We note that some of the results derived in Section 5.4 were based on permissions for which we did not evaluate the accuracy; e.g., `SEND_SMS`, `KILL_BG_PROC`, and `GET_TASKS`. Therefore, the results might have threats to validity. A simple solution to address the concern is to extend the labeled dataset, however, we were not able to perform the additional experiments due to the high cost of labeling descriptions written in two languages. Although not conclusive, we note that we have validated that the descriptions were correctly classified through the manual inspection, using randomly sampled apps; i.e., the obtained results were partially validated.

6. DISCUSSION

In this section, we discuss the feasibility and versatility of the ACODE framework. We also outline several future research directions that are extensions of our work.

6.1 User experience

In this study, we asked participants to read whole sentences carefully, regardless of the size of the text description. In a real-user setting, users might stop reading a text description if it is very long. Studying how the length of text descriptions or the placement of permission-related sentences affect user awareness is a topic for future work. In addition to text descriptions, mobile software distribution platforms provide other information channels, such as meta data or screenshots of an app. As users may also pay attention to these sources of information, studying how these sources provide information about permissions is another research challenge we are planning to address.

6.2 Cost of analysis

Because this work aims to tackle with a huge volume of applications, we adopt light-weight approaches; static code analysis (instead of dynamic code analysis) and keyword-based text analysis (instead of semantic analysis). In the followings, we detail the cost of our approach. The cost of data analysis with the ACODE framework can be divided into two parts: the static code analyzer and the text descriptions analyzer. For the static code analyzer, the most expensive task is the function call analysis because we first need to build function call trees to study whether an API is callable. Our empirical study showed that the task of function call analysis for an application took 6.05 seconds on average. We note that the tasks can be easily parallelized. By parallelizing the tasks with 24 of processes on a commodity PC, we were able to process 200 K apps within a single day. For the text description analyzer, collecting label was the most expensive task. On average, a single participant labeled 1,500 of descriptions within 10 hours. However, once we get the performance evaluation of our approach, we do not need to employ the task again because our work does not need manually-labeled samples. Since we adopt keyword-based approach, analyzing hundred thousands of descriptions was quite fast.

Overall, all the tasks can be completed within a single day, and we can further accelerate the speed if this is desired. As our objective is not to perform the analysis in real-time, we believe that the cost of performing analyses with the ACODE framework is affordable.

6.3 Permissions that should or should not be mentioned.

Android OS manages several permissions with a protection level defined as "dangerous," which means "a higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user [38]." Ideally, users should be aware of all these dangerous permissions. The dangerous permissions can be broadly classified into two categories: for users and for developers. Permissions for users include read/write contacts, access fine location, read/write calendar, read/write user dictionary, camera, microphone, Bluetooth, and send/read SMS. The three resources analyzed in this paper are the permissions aimed at users. Permissions for developers include set debug app, set process limit, signal persistent processes, reorder tasks, write setting, and persistent activity.

Permissions for users are intuitively understandable. Thus, they should be described in the text descriptions. Permissions for developers are difficult for general users to understand; thus, describing them may be confusing. As describing these permissions could even distract users' attention from the text descriptions, they should *not* be mentioned in the text descriptions. For such dangerous permissions aimed at developers, we need to develop another information channel that lets users know about the potential threats in an intuitive way. We note that the ACODE framework can be used to identify dangerous permissions that are least mentioned. Knowledge of such permissions will be useful to develop a new information channel.

7. RELATED WORK

Researchers have studied mobile apps from various viewpoints, including issues of privacy, permission, and user behavior. In this section, we review the previous studies along four axes: system-level protection schemes, large-scale data analyses, user confidence and user behavior, and text descriptions of mobile apps.

7.1 System-level protection schemes

As a means of protecting users from malicious software, several studies have proposed install-time or runtime protection extensions that aim to achieve access control and application isolation mechanisms such as [39, 40, 41, 42, 43]. Kirin [39] performs lightweight certification of applications to mitigate malware at install-time based on a conservative security policy. With regard to install-time permission policies and runtime inter-application communication policies, SAINT [40] provides operational policies to expose the impact of security policies on application functionality, and to manage dependencies between application interfaces. TaintDroid [41] modifies the operating system and conducts dynamic data tainting at runtime in order to track the flow of sensitive data to detect when this data is exfiltrated. Quire [44] is defense mechanisms against privilege escalation attacks with inter-component communication (ICC). Finally, SEAndroid [43] brings flexible mandatory access control (MAC) to Android by enabling the effective use of Security Enhanced Linux (SELinux).

While the above studies improved the system-level security and privacy of smartphone, this work attempts to address the problem from a different perspective – understanding the effectiveness of text description as a potential source of information channel for improving users' awareness of privacy.

7.2 Large-scale data analyses

Several researchers have conducted measurement studies to understand how many mobile apps access to private resources and how they use permissions to do so [2, 3, 4, 5]. A survey report published by Bit9 [2] included a large-scale analysis of Android apps using more than 410,000 of Android apps collected from the official Google Play marketplace. Through the analysis, they revealed that roughly 26% of apps access personal information such as contacts and e-mail, 42% of apps access GPS, and 31% of apps access phone calls or phone numbers. Book et al. [5] analyzed how the behavior of the Android ad library and permissions have changed over time. Through the analysis of 114,000 apps collected from Google Play, they found that the use of most permissions has increased over time, and concluded that permissions required by ad libraries could expose a significant weakness in user privacy and security. From the perspective of dynamic code loading, Poeplau et al.[35] conducted an analysis of 1,632 popular apps, each with more than 1 million installations, and revealed that 9.25% of them are vulnerable to code injection attacks.

7.3 User confidence and user behavior

Several works on user confidence and user behavior discuss users' installation decisions [9, 45, 46, 47]. Refs. [46, 47] studied user behavior in security warnings, and revealed that most users continue through security warnings. Good et al. [45] conducted an ecological study of computer users installing software, and found that providing vague information in EULAs and providing short notices can create an unwarranted impression of increased security. Chin et al. [9] studied security and privacy implications of smartphone user's behaviors based on a set of installation factors, e.g., price, reviews, developer, and privacy. Their study implicates user agreements and privacy policies as the lowest-ranked factors for the privacy. As these studies on user confidence and behavior suggest, user agreements or privacy policies are not effectively informing consumers about privacy issues with apps. Centralized mobile software distribution platforms should provide mechanisms that improve privacy awareness so users can use apps safely and confidently. We believe that our findings obtained using the ACODE framework can be used to complement these studies.

7.4 Text descriptions

As mentioned in section 1, only a few works have focused on text descriptions of mobile apps [10, 11, 12, 13]. The WHYPER framework [11] is the pioneering work that attempted to bridge the semantic gap between application behaviors and user expectations. They applied modern NLP techniques for semantic analysis of text descriptions, and demonstrated that WHYPER can accurately detect text sentences that refer to a permission. Qu et al. [13] indicated an inherent limitation of the WHYPER framework, i.e., the derived semantic information is limited by the use of a fixed vocabulary derived from Android API documents and synonyms of keywords there. To overcome the issue, they proposed the AutoCog framework based on modern NLP techniques extracting semantics from descriptions without using API documents. The key idea behind their approach is to select noun-phrase based governor-dependent pairs related to each permission. They demonstrated that the AutoCog framework moderately improved performance as compared to the WHYPER framework. Gorla et al. [12] proposed the CHABADA framework, which can identify anomalies automatically by applying an unsupervised clustering algorithm to text descriptions and identifying API usage within each cluster. Like our work, CHABADA uses API functions to identify outliers. On the other hand, the aim of ACODE is *not* to find anomalies, but to quantify the effectiveness of text descriptions as a means of making users aware of privacy threats. To this end, using a simple keyword-based approach, the ACODE framework

Table 10: Comparison between related works.

	ACODE	WHYPER	AutoCog	CHABADA	Lin et al. [10]
objective	Understanding inconsistency between codes and descriptions	Identifying sentences that refer to a permission	Assessing description-to-permission fidelity of applications	Identifying outlier apps	Understanding user expectation on sensitive resources
# of apps	200,000	581	83,656	32,308	134
# of studied permissions	11	3	11	N/A	4
markets	Official, Third-party	Official	Official	Official	Official
languages	English, Chinese	English	English	English	English
code analysis	Function call tree analysis	Permission check	Permission check	API analysis	Permission check
description analysis	Keyword-based	Semantic analysis	Semantic analysis	Topic model	N/A

attempts to assess the reasons why text descriptions do not refer to permissions. As we revealed, the performance of our approach is comparable with that of the WHYPER framework. We also note that the ACODE framework is more fine-grained than CHABADA since ACODE checks whether API functions/URIs found in code are callable by employing function call analysis. Finally, Lin et al. [10] studied users' expectations related to sensitive resources and mobile apps by using crowdsourcing. They asked participants to read the provided screenshots and text description of an app, and asked several questions to investigate users' perceptions of the app as related to privacy-sensitive resources. They concluded that users' expectations and the purpose for using sensitive resources have a major impact on users' subjective feelings and their trust decisions. This observation supports the importance of improving users' privacy awareness on mobile software distribution platforms.

We summarize the differences among the above three studies, and our own in table 10. In addition to the technical differences shown above, our work is distinguishable from other studies in its large-scale empirical analysis, which spans across 11 of distinct permissions, two market places, and 200K of text descriptions written in two different natural languages.

8. CONCLUSION

By applying the ACODE framework to 200,000 apps collected from both official and third-party marketplaces, our analysis across the 11 distinct resources revealed four primary factors that are associated with the inconsistencies between text descriptions and use of privacy-sensitive resources: (1) existence of app building services/frameworks that tend to add API permissions/code unnecessarily, (2) existence of prolific developers who publish many applications that unnecessarily install permissions and code, (3) existence of secondary functions that tend to be unmentioned, and (4) existence of third-party libraries that access to the privacy-sensitive resources.

We believe that our work provides an important first step toward improving users' privacy awareness on mobile software distribution platforms. For instance, developers of app building services/frameworks can use our findings to check the behaviour and deployment of their products. Individual mobile app developers can pay attention to our findings when they write text descriptions or use third-party libraries. And mobile software distribution platform providers can pay attentions to all the potential reasons that lead to the inconsistencies between user expectations and developer intentions. Based on the findings revealed by the ACODE framework, they may be able to come up with new information channels that effectively inform users about the use of privacy-sensitive resources.

Acknowledgments

We are grateful to the authors of WHYPER framework [11] for sharing the invaluable datasets with the research community. We also thank Akira Kanaoka for inspiring us to start this work. Our special thanks are to Lira Park, Gracia Rusli, Ahro Oh, Suthinan Thanintranon, Karyu Chen, Xia Tian, Bo Sun, Jiarong Chen, Xuefeng Zhang, Hao Wang, Dan Li, Chen Wang for their assistance in collecting the labeled text descriptions used in this work. Finally, we thank the anonymous reviewers for their thoughtful suggestions for improving this paper. In particular, we thank our shepherd, William Enck for his valuable feedback.

9. REFERENCES

[1] S. Shen and B. Blau, "Forecast: Mobile App Stores, Worldwide, 2013 Update." `https://www.gartner.com/doc/2584918/forecast-mobile-app-stores-worldwide`.

[2] B. Report, "Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks." `https://www.bit9.com/research/pausing-google-play/`.

[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pp. 627–638, 2011.

[4] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[5] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal analysis of android ad library permissions," in *IEEE Mobile Security Technologies (MoST)*, 2013.

[6] Y. Zhou and X. Jiang, "Detecting passive content leaks and pollution in android applications," in *20th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2013.

[7] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012*, May 2012.

[8] Future of Privacy Forum, "FPF Mobile Apps Study." `http://www.futureofprivacy.org/wp-content/uploads/Mobile-Apps-Study-June-2012.pdf`.

[9] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," in *Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[10] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pp. 501–510, 2012.

[11] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *Proceedings of the 22Nd USENIX Conference on Security*, pp. 527–542, Aug 2013.

[12] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*, 2014.

[13] V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

[14] M. P. Lewis, ed., *Ethnologue: Languages of the World*. Dallas, TX, USA: SIL International, seventeenth ed., 2013.

[15] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[16] "Android asset packaging tool." http://www.kandroid.org/guide/developing/tools/aapt.html.

[17] "PScout: Analyzing the Android Permission Specification." `http://pscout.csl.toronto.edu/`.

[18] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pp. 217–228, 2012.

[19] "android-apktool." `http://code.google.com/p/android-apktool/`.

[20] "smali – An assembler/disassembler for Android's dex format." `https://code.google.com/p/smali/`.

[21] "androguard." `https://code.google.com/p/androguard/`.

[22] "Kyoto Text Analysis Toolkit." `http://www.phontron.com/kytea/`.

[23] "Natural Language Toolkit." `http://www.nltk.org`.

[24] "imdict-chinese-analyzer." `https://code.google.com/p/imdict-chinese-analyzer/`.

[25] M. Makrehchi and M. S. Kamel, "Automatic extraction of domain-specific stopwords from labeled documents," in *Proceedings of the IR Research, 30th European Conference on Advances in Information Retrieval*, ECIR'08, pp. 222–233, 2008.

[26] S. E. Robertson and K. S. Jones, "Simple, Proven Approaches to Text Retrieval," Tech. Rep. 356, University of Cambridge Computer Laboratory, 1997.

[27] "Whyper: Towards automating risk assessment of mobile applications." `https://sites.google.com/site/whypermission/`.

[28] "Google play." `http://play.google.com/`.

[29] "Anzhi.com." http://anzhi.com.

[30] "Nduoa market." `http://www.nduoa.com/`.

[31] J. Oberheide and C. Miller, "Dissecting the android bouncer." SummerCon, Brooklyn, NY., 2012. `http://jon.oberheide.org/files/summercon12-bouncer.pdf`.

[32] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pp. 66–77, 2014.

[33] "Official ZXing ("Zebra Crossing") project home." `https://github.com/zxing/zxing`.

[34] "ZBar bar code reader." `http://zbar.sourceforge.net/`.

[35] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[36] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pp. 93–104, 2012.

[37] A. Gianazza, F. Maggi, A. Fattori, L. Cavallaro, and S. Zanero, "Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications," *CoRR*, vol. abs/1402.4826, 2014.

[38] "Android developers guide: App manifest – permission." `http://developer.android.com/guide/topics/manifest/permission-element.html`.

[39] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pp. 235–245, ACM, 2009.

[40] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in android," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, (Washington, DC, USA), pp. 340–349, IEEE Computer Society, 2009.

[41] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[42] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[43] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android," in *NDSS*, The Internet Society, 2013.

[44] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, 2011.

[45] N. Good, R. Dhamija, J. Grossklags, D. Thaw, S. Aronowitz, D. Mulligan, and J. Konstan, "Stopping spyware at the gate: A user study of privacy, notice and spyware," in *Symposium on Usable Privacy and Security (SOUPS)*, pp. 43–52, 2005.

[46] C. Bravo-Lillo, L. F. Cranor, J. Downs, and S. Komanduri, "Bridging the gap in computer security warnings: A mental model approach," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 18–26, 2011.

[47] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *Proceedings of the 22Nd USENIX Conference on Security*, Security'13, 2013.

APPENDIX

Table 11: Top-100 Domain-specific Stop Words: English (top) and Chinese (bottom). The keywords are sorted from the first (top-left) to the last (bottom-right).

(1) :	(2) !	(3) app	(4))	(5) (
(6) -	(7) free	(8) 's	(9) get	(10) feature
(11) time	(12) use	(13) one	(14) new	(15) ?
(16) also	(17) application	(18) game	(19) android	(20) phone
(21) like	(22) make	(23) find	(24) help	(25) ha
(26) version	(27) please	(28) "	(29) "	(30) simple
(31) screen	(32) need	(33) easy	(34) best	(35) play
(36) see	(37) fun	(38) way	(39) want	(40) n't
(41) device	(42) information	(43) many	(44) friend	(45) using
(46) take	(47) know	(48) download	(49) keep	(50) go
(51) u	(52) work	(53) enjoy	(54) ielts	(55) &
(56) full	(57) different	(58) world	(59) show	(60) support
(61) mobile	(62) let	(63) available	(64) 2	(65) level
(66) share	(67) 3	(68) give	(69) day	(70) score
(71) set	(72) 1	(73) check	(74) list	(75) right
(76) number	(77) email	(78) access	(79) great	(80) learn
(81) save	(82) user	(83) note	(84) every	(85) update
(86) well	(87) even	(88) much	(89) button	(90) view
(91) add	(92) smoked	(93) try	(94) live	(95) first
(96) video	(97) search	(98) home	(99) ad	(100) allows

(1) 款	(2) 中	(3) 游戏	(4) 上	(5) 一个
(6) 不	(7) 更	(8) 最	(9) 还	(10) 简单
(11) 会	(12) 都	(13) 人	(14) 手机	(15) 功能
(16) 好	(17) 需要	(18) 更新	(19) 大	(20) 玩
(21) 快	(22) 提供	(23) 试试	(24) 小游戏	(25) 操作
(26) 小	(27) 信息	(28) 非常	(29) 赶快	(30) 软件
(31) 很	(32) 点击	(33) 生活	(34) 时间	(35) 想
(36) 所有	(37) 挑战	(38) 选择	(39) 体验	(40) 各种
(41) 不同	(42) 玩家	(43) 可 爱	(44) 种	(45) 类
(46) 界面	(47) 障碍	(48) 使用	(49) 起来	(50) 看
(51) 支持	(52) 壁纸	(53) 最新	(54) 有趣	(55) 画面
(56) 休闲	(57) 经典	(58) 疯狂	(59) 屏幕	(60) 后
(61) 益智	(62) 完成	(63) 乐趣	(64) 内容	(65) 控制
(66) 帮助	(67) 世界	(68) 应用	(69) 出	(70) 一下
(71) 喜欢	(72) 方式	(73) 下	(74) 收集	(75) 时
(76) 版	(77) 关卡	(78) 进行	(79) 好玩	(80) 美女
(81) 能够	(82) 朋友	(83) nbsp	(84) 用 户	(85) v1
(86) 获得	(87) 一定	(88) 看看	(89) 错过	(90) 消除
(91) 图片	(92) 精美	(93) 能力	(94) 越	(95) 键
(96) 必备	(97) 一起	(98) 任务	(99) 平台	(100) 没有

Table 12: Top-10 Extracted keywords for each permission: English (top) and Chinese (bottom). The keywords are sorted from the first (left) to the last (right).

ACCESS_FINE_LOCATION	(1) gps	(2) location	(3) map	(4) restaurant	(5) direction
	(6) city	(7) event	(8) local	(9) service	(10) area
SEND_SMS	(1) sms	(2) message	(3) sent	(4) send	(5) text
	(6) call	(7) contact	(8) gps	(9) notification	(10) automatically
KILL_BACKGROUND_PROCESSES	(1) task	(2) kill	(3) manager	(4) protection	(5) running
	(6) memory	(7) process	(8) usage	(9) battery	(10) clean
READ_CALENDAR	(1) calendar	(2) reminder	(3) meeting	(4) event	(5) schedule
	(6) widget	(7) google	(8) call	(9) notification	(10) data
CAMERA	(1) camera	(2) scan	(3) photo	(4) event	(5) customer
	(6) direction	(7) restaurant	(8) gallery	(9) offer	(10) community
GET_TASKS	(1) lock	(2) security	(3) task	(4) apps	(5) battery
	(6) password	(7) usage	(8) running	(9) thank	(10) devices
WRITE_SETTINGS	(1) alarm	(2) ring	(3) bluetooth	(4) notification	(5) wifi
	(6) volume	(7) sound	(8) switch	(9) battery	(10) song
RECORD_AUDIO	(1) recording	(2) voice	(3) record	(4) audio	(5) wall
	(6) exclusive	(7) located	(8) direction	(9) client	(10) event
READ_CONTACTS	(1) sms	(2) call	(3) contact	(4) message	(5) text
	(6) send	(7) notification	(8) receive	(9) automatically	(10) service
GET_ACCOUNTS	(1) grab	(2) google	(3) youtube	(4) account	(5) calendar
	(6) contact	(7) feed	(8) fan	(9) join	(10) today
READ_SMS	(1) sms	(2) message	(3) incoming	(4) backup	(5) call
	(6) log	(7) contact	(8) text	(9) notification	(10) data

ACCESS_FINE_LOCATION	(1) GPS	(2) 周边	(3) 附近	(4) 定位	(5) 优惠
	(6) 商家	(7) 酒店	(8) 会 员	(9) 导航	(10) 文
SEND_SMS	(1) 重生	(2) 妻	(3) 宠	(4) 嫁	(5) 总裁
	(6) 妃	(7) 男人	(8) 穿越	(9) 竟然	(10) 世
KILL_BACKGROUND_PROCESSES	(1) 狐	(2) 清理	(3) 进程	(4) 垃圾	(5) 省 电
	(6) 骚扰	(7) 管理器	(8) 耗	(9) 卸载	(10) 内存
READ_CALENDAR	(1) 日程	(2) 日历	(3) widget	(4) 谷歌	(5) 部件
	(6) 优先	(7) 日期	(8) Google	(9) 震动	(10) 彩信
CAMERA	(1) 码	(2) 扫描	(3) 相机	(4) 专用	(5) 淘
	(6) 拍照	(7) 拍 摄	(8) 商品	(9) 购物	(10) 店
GET_TASKS	(1) 结局	(2) 文	(3) 爱情	(4) 爱上	(5) 亲
	(6) 离开	(7) 梦	(8) 幸福	(9) 真的	(10) 终于
WRITE_SETTINGS	(1) 铃声	(2) 总裁	(3) 修	(4) 仙	(5) 夜
	(6) 竟然	(7) 嫁	(8) 女人	(9) 男人	(10) 却
RECORD_AUDIO	(1) 完结	(2) 本文	(3) 结局	(4) 文	(5) 恋
	(6) 语音	(7) 爱情	(8) 爱上	(9) 亲	(10) 离开
READ_CONTACTS	(1) 通话	(2) 来 电	(3) 拨号	(4) 通讯	(5) 短信
	(6) 号码	(7) 打 电话	(8) 备份	(9) 话费	(10) 拨打
GET_ACCOUNTS	(1) Google	(2) 谷歌	(3) Free	(4) 备份	(5) 日历
	(6) 云端	(7) Facebook	(8) 丢失	(9) 百度	(10) 呼叫