Using Text Mining to Infer the Purpose of Permission Use in Mobile Apps

Haoyu Wang[*], **Jason Hong**[†], **Yao Guo**[*]
[*]Key Laboratory of High-Confidence Software Technologies (Ministry of Education),
School of Electronics Engineering and Computer Science, Peking University
[†]Carnegie Mellon University
{howiepku@pku.edu.cn, jasonh@cs.cmu.edu, yaoguo@pku.edu.cn}

ABSTRACT
Understanding the purpose of why sensitive data is used could help improve privacy as well as enable new kinds of access control. In this paper, we introduce a new technique for inferring the purpose of sensitive data usage in the context of Android smartphone apps. We extract multiple kinds of features from decompiled code, focusing on app-specific features and text-based features. These features are then used to train a machine learning classifier. We have evaluated our approach in the context of two sensitive permissions, namely `ACCESS_FINE_LOCATION` and `READ_CONTACT_LIST`, and achieved an accuracy of about 85% and 94% respectively in inferring purposes. We have also found that text-based features alone are highly effective in inferring purposes.

Author Keywords
Permission; purpose; mobile applications; Android; privacy

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION
Mobile apps have seen widespread adoption, with over one million apps in both Google Play and the Apple App Store, and billions of downloads [5, 18]. Mobile apps can make use of the numerous capabilities of a smartphone, including many kinds of sensors (e.g., GPS, camera, and microphone) and a wealth of personal information (e.g., contact lists, emails, photos, and call logs).

Android currently requires developers to declare what permissions an app uses, but offers no mechanisms to specify the purpose of how the sensitive data will be used. Knowing the purpose of a permission request could help with respect to privacy, for example offering end-users more insights as to why an app is using a specific sensitive data. Having a clear purpose of a request could also offer fine-grained access control, for example, disallowing geotagging in sensitive locations (e.g. government or military facilities) while still allowing map searches.

Our specific focus is on developing better methods to infer the purpose of a permission request. Various techniques have been proposed to bridge the semantic gap between users' expectations and app functionality. For example, WHYPER [34] and AutoCog [40] apply natural language processing techniques to an app's description to infer permission use. CHABADA [19] clusters apps by their descriptions to identify outliers in each cluster with respect to the API usage. RiskMon [23] builds a risk assessment baseline for each user according to the user's expectations and runtime behaviors of trusted applications, which can be used to assess the risks of sensitive information use and rank apps. Amini *et al.* introduced Gort [3], a tool that combines crowdsourcing and dynamic analysis, which could help users understand and flag unusual behaviors of apps.

Our research thrust is closest to Lin *et al.* [26, 27], who introduced the idea of inferring the purpose of a permission by analyzing what third-party libraries an app uses. For example, if a location data is only used by an advertising library, then they can infer that it is used for advertising. Lin *et al.* [27] categorized the purposes of several hundred third-party libraries (advertising, analytics, social network, etc), used crowdsourcing to ascertain people's level of concern for data use (e.g. location for advertising versus location for social networking), and clustered and analyzed apps based on their similarity.

In this paper, we evaluate the effectiveness of using text analysis techniques on decompiled code for inferring the purpose of a permission use. We focus on custom written code as opposed to third-party libraries, though our techniques should also work for libraries. One of the core insights underlying our work is that, unless an app has been obfuscated, compiled Java class files still retain the text of many identifiers, such as class names, method names, and field names. These strings offer a hint as to what the code is doing. As a simple example, if we find custom code that uses the location permission and has method or variable

UbiComp '15, September 7-11, 2015, Osaka, Japan.
Copyright 2015 © ACM 978-1-4503-3574-4/15/09...$15.00.
http://dx.doi.org/10.1145/2750858.2805833

names such as "photo", "exif", or "tag", it is likely that it uses location data for the purpose of "geotagging".

In our approach, we first decompile apps and search the decompiled code to determine where sensitive permissions are used. Then we extract multiple kinds of features from the decompiled code, including both *app-specific features* (e.g., API calls, the use of Intent and Content Provider) and *text-based features* (TF-IDF results of meaningful words extracted from package names, class names, interface names, method names, and field names). We use these features to train a classification model for inferring the purpose of permission uses.

We created a taxonomy for purposes on how apps use two sensitive permissions, namely `ACCESS_FINE_LOCATION` (*location* for short) and `READ_CONTACTS` (*contacts* for short). We use this taxonomy to manually examine and label the behavior of 460 instances using location (extracted from 305 apps), and 560 instances using contacts (extracted from 317 apps). Here, an instance is defined as a directory of source code, thus a single app may yield more than one instance. We use this data to train a machine learning classifier. Using 10-fold cross validation, experiments show that we can achieve about 85% accuracy in inferring the purpose of location use, and 94% for contact list use.

This paper makes the following research contributions:

- We introduce the idea of using text analysis and machine learning techniques on decompiled code to infer the purpose of permission uses. To the best of our knowledge, our work is the first attempt to infer purposes for custom written code (as opposed to third-party libraries or app descriptions).
- We evaluate our approach for two frequently-used permissions (location and contact lists) on 1020 labeled instances (permission-related directories) that belong to 622 different apps. The results of 10-fold cross validation show that we can achieve a purpose inference accuracy of about 85% for location and 94% for contacts.
- We investigate the effectiveness of different kinds of features extracted from decompiled Android apps, showing that text-based features offer a very high gain, with other features offering marginal improvements.

BACKGROUND AND RELATED WORK

Our study is mainly related to three bodies of work: research on detecting where permissions are used in code; research on detecting and bridging the semantic gap between users' expectations and app functionalities; and inferring the purpose of permission uses.

Detecting Where Permissions Are Used in Code

One challenge in our work is determining what parts of an app use sensitive permissions, so that we know where to apply feature extraction. There is a body of work that solve a different problem, namely determining the precise set of permissions an app needs, which we leverage in our work.

Android uses install-time permissions to control access to system resources. Permissions are declared in manifest files by developers, and users must accept this list of permissions at install time. However, not all the permissions an app requests are essential for it to run properly. Past studies have found that some apps request more permissions than they actually use, which is called *permission overprivilege*. One recent work [50] showed that more than 85% of the apps pre-installed on vendor customized smartphones suffer from permission overprivilege. According to another study [8], the permission gap could be leveraged by malware to achieve malicious goals in several ways, such as code injection and return-oriented programming.

Various approaches [16, 6, 8, 9] have been proposed to determine the precise set of permissions that an app needs. These approaches typically involve first building a permission map that identifies what permissions are needed for each API call, then performing static analysis to identify permission-related API calls.

For example, STOWAWAY [16] uses automated testing techniques to generate unit test cases for API calls, Content Providers, and Intents. It also instruments the code points in Android where permissions are checked, to log permission checks and generate the permission map. PScout [6] and COPES [8, 9] perform call-graph based analysis on the Android framework to build the permission map. Permission specifications are generated for API calls that can be reached from a permission check.

Our work uses the permission map built by PScout [6] to identify which permissions are actually used in the code and where they are used. Then we extract features to infer the purpose of the permission-related code.

The Gap Between User Expectations and App Behaviors

Past studies [17, 11, 14] have shown that mobile users have a poor understanding of permissions. They cannot correctly understand the permissions they grant, while current permission warnings are not effective in helping users make security decisions. Meanwhile, users are usually unaware of the data collected by mobile apps [17, 43]. Several studies [2, 20, 25] have been proposed to focus on raising users' awareness of the data collected by apps, informing them of potential risks and help them make decisions.

Furthermore, previous studies [7, 24] suggested that there is a semantic gap between users' expectations and app behaviors. Recent research [34, 40, 48, 19, 23, 51, 3] has looked at ways to incorporate users' expectations to assess the use of sensitive information, proposing new techniques to bridge the semantic gap between users' expectations and app functionalities.

For example, WHYPER [34] and AutoCog [40] use natural language processing (NLP) techniques to infer permission use from app descriptions. They build a permission semantic model to determine which sentences in the description indicate the use of permissions. By comparing the result with the requested permissions, they can detect inconsistencies between the description and requested permissions. ASPG [48] has proposed generating semantic permissions using NLP techniques on the app descriptions.

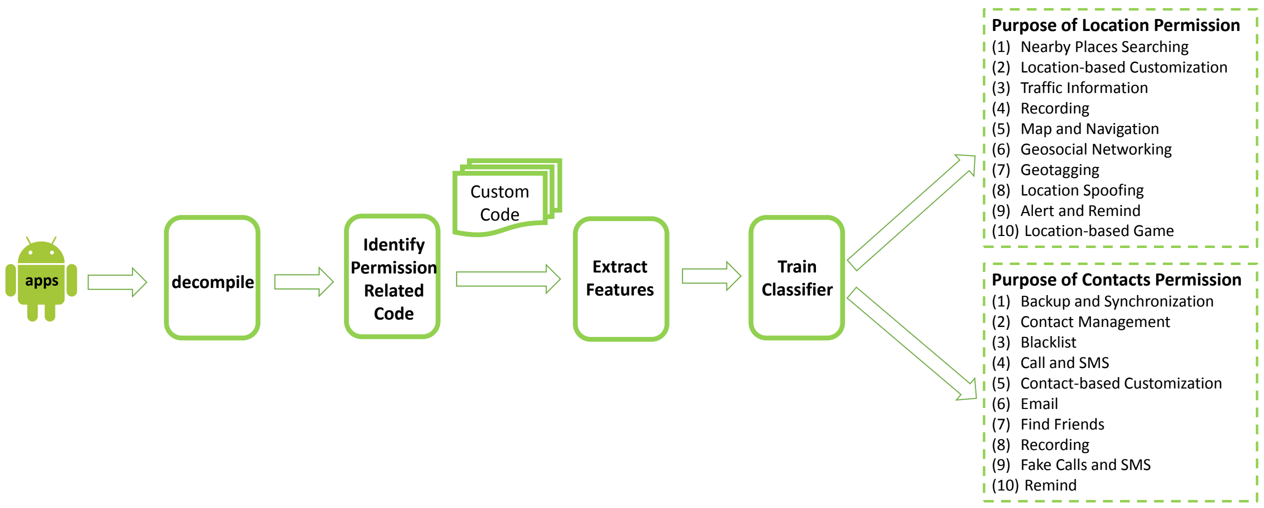

Figure 1. The overall architecture of our approach. We first decompile each app and filter out third-party libraries using a list of the most popular libraries. We then use static analysis to identify where permission-related code is located. We extract several kinds of features from this code and then train the classifier. The classifier outputs 10 different purposes for location and for contacts.

It then tailored the requested permissions that are not listed in the semantic permissions to get the minimum set of permissions an app needs. CHABADA [19] uses Latent Dirichlet Allocation (LDA) on app descriptions to identify the main topics of each app, and then clusters apps based on related topics. By extracting sensitive APIs used for each app, it can identify outliers which use APIs that are uncommon for that cluster. All of these approaches have attempted to infer permission use or semantic information from app description, and bridge the gap between app description and functionalities.

Ismail *et al.* [21] have leveraged crowdsourcing to find the minimal set of permissions to preserve the usability of the app for diverse users. RiskMon [23] builds a risk assessment baseline for each user according to the user's expectations and runtime behaviors of trusted applications, which can be used to assess the risks of sensitive information use and rank apps. Amini *et al.* have introduced Gort [3], a tool that combines crowdsourcing and dynamic analysis to help help users understand and flag unusual behaviors of apps. AppIntent [51] uses symbolic execution to infer whether a transmission of sensitive data is by user intention or not. Past research [42, 30, 47] has also measured users' privacy preferences in different contexts. For example, Shih *et al.* [42] found that the purpose of data access is the main factor affecting users' choices.

Our work contributes to this body of knowledge, looking primarily at using text mining technique on decompiled code to infer the purpose of permission uses. Our approach is also static, so we do not have to run an app, offering good potential for scalability.

Determining the Purpose of Permission Uses

Lin *et al.* [26, 27] first introduced the idea of inferring the purpose of a permission request by analyzing what third-party libraries an app uses. They categorized the purposes of 400 third-party libraries (advertising, analytics, social network, etc.), and used crowdsourcing to ascertain people's level of concern for data use (e.g. location for advertising versus location for social networking). Then they clustered and analyzed apps by similarity. Their results suggest that both users' expectations and the purpose of permission use have a strong impact on users' subjective feelings and their mental models of mobile privacy.

However, a major gap in this existing work is how to infer the purpose of a permission request in custom-written code, which turns out to be a much more difficult problem. According to the results of a recent work [1] that analyzed 1.2 million apps from Google Play, most permission requests occur in custom code. Specifically, for apps that use the location permission, more than 55.7% of them use the location permission in their custom code. For apps that use the contacts permission, more than 71.2% of them use the contacts permission in their custom code.

Our work focuses on addressing this gap to infer the purpose of permission uses in custom code, relying primarily on text mining and machine learning techniques. As such, while our overall goal is similar to past work, we focus on a different part of the problem space and use very different techniques.

INFERRING PURPOSES

Overview

As shown in Figure 1, we first use static analysis to identify parts of custom code that uses location or contacts permission. Then, we extract various kinds of features from the custom code using text-mining (e.g., splitting identifier names and extracting meaningful text features) and static analysis (identifying important APIs, Intents and Content Providers). In the training phase, we manually label instances to train a classifier. The classifier outputs the purpose of an instance as one of ten different purposes for `location` or one of ten different purposes for `contacts`. Note that we opted not to examine third-party libraries here, partly because there was no previous work for custom code,

Table 1. The purpose of the location permission uses in custom code.

Purpose	Description	#Instances	Example Apps
Search Nearby Places	Find nearby hotels, restaurants, bus stations, bars, pharmacies, hospitals, etc.	50	Booking
Location-based Customization	Provide news, weather, time, activities information based on current location	50	Weather Channel
Transportation Information	Taxi ordering, real-time bus and metro information, user-reported bus/metro location	50	Easy Taxi
Recording	Real-time walk/run tracking, location logging and location history recording, children tracking	50	RunKeeper
Map and Navigation	Driving route planning and navigation	50	OsmAnd
Geosocial Networking	Find nearby people/friends, social networking check-in	50	Badoo
Geotagging	Add geographical identification metadata to various media such as photos and videos	30	TagMe
Location Spoofing	Sets up fake GPS location	30	Fake GPS
Alert and Remind	Remind location-based tasks, disaster alert such as earthquake	50	GeoTask
Location-based game	Games in which the gameplay evolves and progresses based on a player's location	50	Tidy City

Table 2. The purpose of the contacts permission uses in custom code.

Purpose	Description	#Instances	Example Apps
Backup and Synchronization	Backup contacts to the server, restore and sync contacts	61	MCBackup
Contact Management	Remove invalid contacts, delete/merge duplicate contacts	30	Naver Contact
Blacklist	Block unwanted calls and SMS	52	EasyBlacklist
Call and SMS	Make VoIP/Wifi calls using Internet, send text message	54	Viber
Contact-based Customization	Add contacts to a custom dictionary for input methods, change ringtone and background based on contacts	51	Sogou Input Method
Email	Send Email to contacts	78	Gmail
Find friends	Add friends from contacts, find friends who use the app in contact list	46	Instagram
Record	Call Recorder, call log and history	93	Call Recorder
Fake Calls and SMS	Select a caller from contact list and give yourself a fake call or SMS to get out of awkward situations	49	Fake Caller
Remind	Missed call notification, remind you to call someone	46	WillCall

and partly because we found that many third-party libraries were obfuscated, which makes static analysis and text mining more difficult.

We focus on inferring the purpose for two sensitive permissions: `location` and `contacts`. We created a taxonomy of the purpose of location permission use and the purpose of contacts permission use, as shown in Table 1 and Table 2. Note that we did not find any purposes such as "advertisement" or "analytics" in custom code, because apps usually use third-party libraries for these functionalities.

Decompiling Apps

For each app, we first decompile it from DEX (Dalvik Executable) into intermediate *Smali* code using Apktool [4]. Smali is a kind of register-based language, and one Smali file has exactly one corresponding Java file. We use this format because we found that it is easier to identify permission-related code based on this format.

We then decompile each app to *Java* source code using dex2jar [13] and JD-Core-Java [22]. We use this decompiled Java source code to extract features. Previous research [15] found that more than 94% of classes could be successfully decompiled. One potential issue, though, is that DEX can be obfuscated. In practice, we found that roughly 10% of the apps are obfuscated in our experiment. We will discuss this issue further in the *Discussion* section.

Because our work focuses on custom code, we first filter third-party libraries before we identify the permission-related code and extract features. We use a list of several hundred of third-party libraries built by past work [26] to remove libraries. While this list is not complete, it works reasonably well in practice.

Identifying Permission-Related Code

For Android apps, three types of operations are permission-related: (1) explicit calls to standard Android APIs that lead to the $checkPermission$ method, (2) methods involving sending/receiving Intents, and (3) methods involving management of Content Providers.

We leverage the permission mapping [35] provided by PScout [6] to determine which permissions are actually used in the code and where they are used. More specifically, we

Table 3. The features used in classification model.

Type	Kind	Features Description	Counts	Method
App Specific Features	Android API	Call frequency of each permission-related API	A 680 dimension vector, each value represents the number of occurrences of corresponding API.	Static Analysis
	Android Intent	Call frequency of each permission-related Intent	A 97 dimension vector, each value represents the number of occurrences of corresponding Intent	
	Content Provider	Call frequency of each permission-related Content Provider Uri	A 78 dimension vector, each value represents the number of occurrences of corresponding Content Provider	
Text-based Features	Package-level Features	Key words extracted from current package names	Calculate TF-IDF for all the key words, with each instance represented as a TF-IDF vector	Text Mining
	Class-level Features	Key words extracted from class and interface names		
	Method-level Features	Key words extracted from defined and used method and parameter names		
	Variable-level Features	Key words extracted from defined and used variable names		

created a lightweight analyzer for searching for sensitive API invocations, Intents, and Content Providers in Smali code. For example, if we find the Android API string "`Landroid/location/LocationManager;->get LastKnownLocation`" in the code, we know it uses the `location` permission. Since Smali code preserves the original Java package structure and has a one-to-one mapping with Java code, we can pinpoint which decompiled source file uses a given permission.

Code Granularity for Inferring Purposes

An important question here is: what is the granularity of code that should be analyzed? One option is to simply analyze the entire app; however, this is not feasible since an app might use the same permission for several purposes in different places. For example, the same app might use `location` for geo-tagging, nearby searches, and advertisement, but a coarse-grained approach might not find all of these purposes. Another option is a fine-grained approach, such as at the method level or class level. However, in our early experiments, we found that there was often not enough meaningful text information contained in a single method or class, making it hard to infer the purpose.

In our work, we decided to use all of the classes in the same `directory` as our level of granularity. In Java, a directory (or file folder) very often maps directly to a single package, though for simplicity we chose to use directories rather than packages. Conceptually, a directory should contain a set of classes that are functionally cohesive, in terms of having a similar goal. We assume that a directory will also only have a single purpose for a given permission, which we believe is a reasonable starting point. Thus, we use static analysis to identify all the directories that use a given sensitive permission, and then analyze each of those directories separately. Note that we only consider the classes in a directory, without considering code in subdirectories.

Feature Extraction

A number of features are used for inferring different kinds of purposes. We group the features into two categories: *app-specific features* and *text-based features*, as shown in Table 3. App-specific features are based on app behaviors and code functionality, while text-based features rely on meaningful identifier names as given by developers.

App-Specific Features

App-specific features include permission-related APIs, Intents and Content Providers. We use these features since they should, intuitively, be highly related to app behaviors. For example, for the contacts permission, we find that API "`sendTextMessage()`" is often used for the "Call and SMS" purpose, but very rarely so for other purposes.

We use static analysis to extract these features. For each kind of API, Intent, and Content Provider, the feature is represented by the number of calls (rather than a binary value of whether the API was used at all), allowing us to consider weights for different features. We scale these features to [0, 1] before feeding them to the classifier. Features with higher values mean they are used more in the code than features with lower values.

Due to the large number of APIs in Android (more than 300,000 APIs according to previous research [6]), it is not feasible to take all of them as features, thus we choose to use *documented permission-related APIs*. Besides, we also use *permission-related Intents* and *permission-related Content Providers* as features. For Android 4.1.1, there are a total of 680 kinds of documented permission-related APIs [37], 97 kinds of Intents associated with permissions [39], and 78 kinds of Content Provider URI Strings associated with permissions [38]. In total, we use 855 kinds of app-specific features. We represent each instance as a feature vector, with each item in the vector recording the number of occurrences of the corresponding API, Intent or Content Provider.

(1) Permission-related APIs

This set of features are related to APIs that require an Android permission. During our experiment, we found that some distinctive APIs could be used to differentiate purposes. For example, some Android APIs in the package "`com.android.email.activity`" are related to contacts permission, and they are often used for "email" purpose. Thus for instances that use such APIs, it is quite possible that it uses contacts for "email" purpose.

We use a list of 680 documented APIs that correlate to 51 permissions provided by Pscout [37], and search for API strings such as "`requestLocationUpdates`" in the decompiled code. Each instance corresponds to a 680 dimension vector, while each item in the vector represents the number of occurrences of the corresponding API.

(2) Intent and Content Providers

We also extract features related to permission-related Intent and Content Provider invocations. Intents can launch other activities, communicate with background services, and interact with smartphone hardware. Content Providers manage access to a structured set of data. For example, Intents such as "`SMS_RECEIVED`" and Content Providers such as "`content://sms`" mostly appear in instances with the "Call and SMS" purpose.

We use a list of 97 Intent [39] and 78 Content Provider URI strings [38]. We search for Android Intent strings such as "`android.provider.Telephony.SMS_RECEIVED`" and Content Provider URI strings such as "`content://com.android.contacts`" in the decompiled code. Each instance corresponds to a 97 dimension Intent feature vector and a 78 dimension Content Provider feature vector, respectively. Each item in the vector represents the number of occurrence of the corresponding Intent or Content Provider.

Text-based Features

We extract text-based features from various identifiers in decompiled Java code. Package names, class names, method names, and field names (instance variables, class variables, and constants) are preserved when compiling, though local variables and parameter names are not. Our goal here is to extract meaningful key words from these names as features.

However, there are several challenges in extracting these features. First, naming conventions may vary widely across developers. Second, identifiers in decompiled Java code are not always words. For example, the method name "findRestaurant" cannot be used as a feature directly. Rather, we want the embedded word "restaurant". Thus, we need to split identifiers appropriately to extract relevant words. Third, not all words are equally useful, and we should consider weights for different words.

We extract text-based features as follows. First, we apply heuristics to split identifiers into separate words. Then we filter out stop words to eliminate words that likely offer little meaning. Next, the remaining words are stemmed into their respective common roots. Finally, we calculate the TF-IDF vector of words for each instance.

(1) Splitting Identifiers

We use two heuristics to split identifiers, namely *explicit patterns* and a *directory-based approach*.

By convention, identifiers in Java are often written in *camelcase*, though underscores are sometimes used. For identifiers with explicit delimitations, we use their construction patterns to split them into subwords. The identifier patterns we used are as listed as follows:

Algorithm 1 Dictionary-based Identifier Splitting Algorithm

```
Input: $identifierI$ and $wordlist$
Output: a list of splitted $keywords$
1: initial $keywords$ = NULL
2: $subword \leftarrow FindLongestWord(I, wordlist)$
3: while $subword \neq NULL$ and $len(I) > 0$ do
4:     keywords.add(subword)
5:     if $len(I) = len(subword)$ then
6:         break
7:     end if
8:     $I \leftarrow identifier.substring(len(subword), len(I))$
9:     $subword \leftarrow FindLongestWord(I, wordlist)$
10: end while
```

$camelcase(1) : AbcDef \rightarrow Abc, Def$

$camelcase(2) : AbcDEF \rightarrow Abc, DEF$

$camelcase(3) : abcDef \rightarrow abc, Def$

$camelcase(4) : abcDEF \rightarrow abc, DEF$

$camelcase(5) : ABCDef \rightarrow ABC, Def$

$underscore : ABC\_def \rightarrow ABC, def$

However, some identifiers do not have clear construction patterns. In these cases, we use a dictionary-based approach to split identifiers. We also use this dictionary to split subwords extracted in the previous step. We use the English wordlist provided by Lawler [49]. We also add some domain-related and representative words into the list, such as `Wifi`, `jpeg`, `exif`, `facebook`, `SMS` etc. For each identifier, we find the longest subword from the beginning of the identifier that can be found in the wordlist. Details of the algorithm are shown in Algorithm 1.

(2) Filtering

We then build a list to filter out stop-words. In addition to common English words, we also filter out words common in Java such as "set" and "get", as well as special Java keywords and types, such as "public", "string", and "float".

(3) Stemming

Stemming is a common Natural Language Processing technique to identify the "root" of a word. For example, we want both singular forms and plural forms, such as "hotel" and "hotels", to be combined. We use the Porter stemming algorithm [36] for stemming all words into a common root.

(4) TF-IDF

After words are extracted and stemmed, we use TF-IDF to score the importance of each word for each instance. TF-IDF is good for identifying important words in an instance, thus providing great support for the classification algorithm. Common words that appear in many instances would be scaled down, while words that appear frequently in a single instance are scaled up. To calculate TF, we count the number of times each word occurs in a given instance. IDF is calculated based on a total of 7,923 decompiled apps.

Classification Model

Since the ranges of feature values vary widely, we normalize them by scaling them to [0, 1]. Then we apply machine learning techniques to train a classifier. We have evaluated three different algorithms for the classification: SVM [46], Maximum Entropy [31], and C4.5 Decision Tree [10]. The implementation of SVM is based on the python scikit-learn [41] package. We use a SVM with linear kernel, and the parameter C is set as 1 based on our practice. Maximum entropy and C4.5 algorithms are based on Mallet [29]. We compare different classifiers using various metrics.

EVALUATION

Dataset

We have downloaded 7,923 apps from Google Play in December 2014, all of which were top-ranked apps across 27 different categories. For text-based features, we calculate IDF based on a corpus of these apps.

To train the classifier, we use a supervised learning [45] approach, which requires labeled instances. We focus on apps that used location or contacts permissions. After decompiling the apps and filtering out third-party libraries, we use static analysis to identify permission-related custom code. Each directory of code that uses location or contacts permission is an instance.

To facilitate accurate classifications, we try to manually label at least 50 instances for each purpose. However, for the location permission, there are more than 3000 instances in our dataset, so we stop once we get more than 50 examples for a given purpose. As shown in Table 1, we have 50 labeled instances for most of the purposes, except for some purposes that have fewer instances in our dataset (we labeled 30 instances for "geotagging" and "location spoofing" purposes). In contrast, for contacts permission, we have found only fewer than 800 instances in our dataset, so we manually checked and labeled the purposes for all of these instances (which is why the number of instances in Table 2 are not as uniform as those in Table 1).

Labeling Purposes

To label the purpose of an instance, we manually inspect the decompiled code, especially the methods and classes that use location or contacts permission. We examine the method and variable names, as well as the parameters and sensitive APIs used in methods to label purposes. For example, in one case, we found custom code using location data and having method and variable names containing "temperature" and "wind", which we labeled as "location-based customization". As another example, we found another instance using photo files and location information (longitude and latitude) by calling the API "`getLastKnownLocation()`", which we labeled as "geotagging". As a third example, we saw an instance invoked API "`sendTextMessage()`" after getting contacts, which we labeled as "Call and SMS" purpose. These examples convey the intuition behind how we label instances and why we these features for the machine learning algorithms.

We also looked at the app descriptions from Google Play to help us label purposes. However, for most of the apps we examined, we could not find any indication of the purpose of permission use. This finding matches previously reported results [40], which found that for more than 90% of apps, users could not understand why permissions are used based solely on descriptions. It indicates the importance of inferring the purpose of permission uses, which could offer end-users more insight as to why an app is using sensitive data.

In total, we manually labeled the purposes of 1,020 instances that belong to 622 different apps, with 460 instances for *location* and 560 instances for *contacts*. Each purpose has 30 to 90 instances, which is shown in Table 1 and Table 2.

Note that our dataset is not comprehensive. For a few apps, we could not understand how permissions are used: thus we did not include them. Our dataset also does not include some apps that have unusual design patterns for using sensitive data. We will offer more details on this issue in the discussion. However, these cases only account for a small portion of our dataset. We feel that our data set is good enough as an initial demonstration of our idea.

Evaluation Method

We used 10-fold cross validation [12] to evaluate the performance of different classifiers. That is, we split our dataset 10 times into 10 different sets for training (90% of the dataset) and testing (10% of the dataset). We manually split our dataset into 10 different sets to ensure that instances of each purpose are equally divided, and that there was no overlap between training and test sets across cross-validation runs. To evaluate the performance of different classifiers, we present metrics for each classification label and metrics for the overall classifier.

Evaluation Metrics

For each class, we measure the number of true positives (*TP*), false positives (*FP*), true negatives (*TN*), and false negatives (*FN*). We also present our results in terms of *precision*, *recall*, and *f-measure*. Precision is defined as the ratio of the number of true positives to the total number of items reported to be true. Recall is the ratio of the number of true positives to the total number of items that are true. F-measure is the harmonic mean of precision and recall.

To measure the overall correctness of the classifier, we use the standard metric of *accuracy* as well as *micro-averaged* and *macro-averaged* metrics [32, 33] to measure the precision and recall. For micro-averaged metrics, we first sum up the *TP*, *FP*, *FN* for all the classes, and then calculate precision and recall using these sums. In contrast, macro-averaged scores are calculated by first calculating precision and recall for each class and then taking the average of them. Micro-averaging is an average over instances, and so classes that have many instances are given more importance. In contrast, macro-averaging gives equal weight to every class. We calculate micro-averaged precision, micro-averaged recall, macro-averaged precision and macro-averaged recall as follows, where c is the number of different classes.

Table 4. The results of inferring the purpose of location uses.

Classification Algorithm	Accuracy	Macro-average Precision	Macro-average Recall
SVM	81.74%	85.51%	83.20%
Maximum Entropy	**85.00**%	**87.07**%	**85.88**%
C4.5	79.57%	83.26%	81.77%

Table 5. The results of inferring the purpose of location permission uses for each category (Maximum Entropy).

Purpose	Precision*	Recall*	F-measure*
L1 Search Nearby Places	76.85%	84.58%	78.99%
L2 Location-based Customization	96.67%	96.33%	95.98%
L3 Transportation Information	100%	86.81%	92.02%
L4 Recording	80.33%	79.19%	77.04%
L5 Map and Navigation	80.54%	93.85%	84.15%
L6 Geosocial Networking	82.57%	87.31%	83.66%
L7 Geotagging	100%	77.67%	84.39%
L8 Location Spoofing	75.48%	90.00%	80.42%
L9 Alert and Remind	100%	76.63%	85.40%
L10 Location-based Game	80.50%	86.38%	81.48%

* The results of precision, recall and f-measure are mean values of 10-fold cross validation.

$$MicroAvg_{Precision} = \frac{\sum_{i=1}^{c} TPi}{\sum_{i=1}^{c} TPi + \sum_{i=1}^{c} FPi} \quad (1)$$

$$MicroAvg_{Recall} = \frac{\sum_{i=1}^{c} TPi}{\sum_{i=1}^{c} TPi + \sum_{i=1}^{c} FNi} \quad (2)$$

$$MacroAvg_{Precision} = \frac{\sum_{i=1}^{c} Precision_i}{c} \quad (3)$$

$$MacroAvg_{Recall} = \frac{\sum_{i=1}^{c} Recall_i}{c} \quad (4)$$

Note that both micro-averaged precision and micro-averaged recall are equal to the *accuracy* of the classifier in our experiment. Thus, we only list the *accuracy* and *macro-averaged metrics* in Table 4 and Table 7.

Results of Inferring Location Purposes

Our results in classifying the purpose of *location* is shown in Table 4. The Maximum Entropy algorithm performs the best, with an overall accuracy of 85%. The results of SVM and C4.5 algorithms also perform reasonably well.

Table 5 presents more detailed results for each specific purpose. The results across different categories vary greatly. The category "location-based customization" achieves the best result, with precision and recall both higher than 96%. The categories "search nearby places" and "location spoofing" have the lowest precision, both under 80%. The purposes "geotagging" and "alert and remind" have 100% precision, but recall under 80%.

Table 6. The confusion matrix of inferring the purpose of location permission use (Maximum Entropy). The purpose number corresponds to that listed in Table 5. Each value is the sum of 10-fold cross validation. Each column represents the instances in a predicted class, while each row represents the instances in an actual class.

Label	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	Total
L1	42	-	-	-	2	1	-	3	-	2	50
L2	1	48	-	-	-	-	-	1	-	-	50
L3	2	-	44	-	1	1	-	1	-	1	50
L4	3	-	-	38	2	3	-	3	-	1	50
L5	-	1	-	1	46	-	-	-	-	2	50
L6	4	-	-	2	-	43	-	-	-	1	50
L7	-	-	-	4	3	-	21	2	-	-	30
L8	-	-	-	-	2	-	-	28	-	-	30
L9	1	-	-	2	1	2	-	2	39	3	50
L10	3	-	-	2	1	2	-	-	-	42	50
Total	56	49	44	49	58	52	21	40	39	52	460

Table 7. The results of inferring the purpose of contacts permission uses

Classification Algorithm	Accuracy	Macro-average Precision	Macro-average Recall
SVM	93.94%	94.38%	92.94%
Maximum Entropy	**94.64**%	**94.42**%	**93.96**%
C4.5	92.86%	91.36%	89.59%

Table 6 shows more details about misclassifications. The category "search nearby places" has the most false positives (see column L1, 14 of 56 classified instances), and 4 misclassified instances belong to "geosocial networking" category. The category "recording" has the most false negatives (see row L4, 12 of 50 labeled instances), and most of them are misclassified as "search nearby places", "geosocial networking" and "location spoofing".

Results of Inferring Contacts Purposes

Our results for inferring the purpose of contacts is shown in Table 7. All three classification algorithms have achieved better than 90% accuracy, with the Maximum Entropy classifier still performing the best at 94.64%.

Table 8 presents details on each category. Our results show that we can achieve high precision and recall for most categories, especially "contact-based customization", "record" and "fake calls and SMS", which have both the precision and recall higher than 95%. However, "contact management" category is not as good, with both precision and recall under 85%.

Table 9 shows the confusion matrix. The category "call and SMS" has the most false positives (see column C4, 9 of 61 classified instances), and "find friends" has the most false negatives (see row C7, 6 of 46 labeled instances). Three instances that belong to "find friends" category are misclassified as "call and SMS" purpose.

Qualitative Analysis of Classification Results

Here, we examine why some categories performed well, while others did not. We inspected several instances and found two factors that play important role in the classification: *distinctive features* and *the number of features*.

Table 8. The results of inferring the purpose of contacts permission use for each category (Maximum Entropy).

Purpose	Precision*	Recall*	F-measure*
C1 Backup and Synchronization	98.75%	94.92%	96.52%
C2 Contact Management	84.33%	84.17%	81.83%
C3 Blacklist	94.17%	93.14%	92.81%
C4 Call and SMS	84.58%	97.08%	89.56%
C5 Contact-based Customization	98.75%	98.33%	98.42%
C6 Email	94.87%	97.09%	95.77%
C7 Find Friends	93.50%	84.17%	87.06%
C8 Record	96.87%	100%	98.35%
C9 Fake Calls and SMS	98.33%	96.67%	97.42%
C10 Remind	100%	94.07%	96.69%

* The results of precision, recall and f-measure are mean values of 10-fold cross validation.

Table 9. The confusion matrix of inferring the purpose of contacts permission use (Maximum Entropy). The purpose number corresponds to that listed in Table 8. Each value is the sum of 10-fold cross validation. Each column represents the instances in a predicted class, while each row represents the instances in an actual class.

Label	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Total
C1	57	1	-	1	-	1	1	-	-	-	61
C2	1	25	-	1	-	-	2	1	-	-	30
C3	-	-	48	1	-	2	1	-	-	-	50
C4	-	1	1	52	-	-	-	-	-	-	54
C5	-	-	-	-	50	-	-	1	-	-	51
C6	-	2	-	1	-	75	-	-	-	-	78
C7	-	-	1	3	1	1	40	-	-	-	46
C8	-	-	-	-	-	-	-	93	-	-	93
C9	-	1	-	-	-	-	-	1	47	-	49
C10	-	-	-	2	-	-	-	-	1	43	46
Total	58	30	50	61	51	79	44	96	48	43	560

Categories with high precision and recall tend to have distinctive features. For example, instances in "location-based customization" had words like "weather", "temperature" and "wind", which were very rare in other categories. In contrast, misclassified instances had more generic words. For example, the labeled instance "`com.etech.placesnearme`" used location information to search nearby places, and its top key words were "local", "search", "place", and "find" etc., which also frequently appeared in other categories. In our experiment, it was misclassified as the "geosocial networking" purpose.

On the other hand, most misclassified instances have fewer features, meaning that there is less meaningful text information that we could extract. For example, "`com.flashlight.lite.gps.passive`" uses location information for "recording". However, it only has 19 kinds of word features and 6 kinds of API features, which is far less than other instances that have hundreds of features. This instance was misclassified as "map and navigation" category in our experiment.

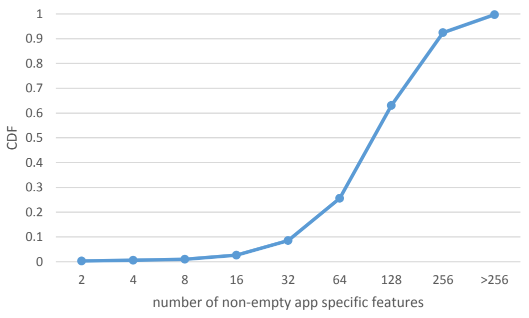

Figure 2. The distribution of the number of non-empty app-specific features per instance

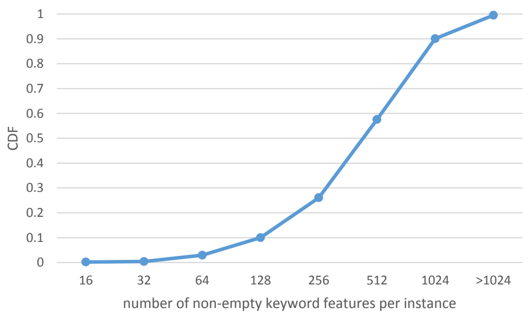

Figure 3. The distribution of the number of non-empty text-based features per instance

Feature Comparison

We are also curious how well text-based features alone are able to perform in the process, since that is the most novel aspect of our work. We train our classifiers using text-based features only and compare the results against classifiers trained by both text-based and app-specific features. The results are shown in Table 10.

We can see that text-based features alone can achieve an accuracy of 81.97% and 93.57% for location and contacts permissions respectively. Incorporating all the features, the performance has only 1.07% to 4.22% improvement. This result suggests text-based features are quite good, while app-specific features play a supporting role.

Figures 2 and 3 offer one possible explanation. These figures show the number of non-empty app-specific features and non-empty text-based features for each instance. We can see that instances almost always have more text-based features than app-specific features, which may be the main reason why text-based features play the leading role in the classifier. The number of text-based features for each instance is about 4 times higher than the number of app-specific features on average (270 and 62 respectively). More than 90% of the instances have fewer than 256 kinds of app-specific features, and in particular, 3% of them have only fewer than 16 kinds of app-specific features. In contrast, more than 74% of the instances have over 256 text-based features, and roughly 10% have over 1024.

One possible implication, and an area of future work, is to develop more app-specific features that can help capture the essence of how sensitive data is used.

Table 10. Using text-based features vs using all features. Text-based features achieve very good accuracy alone, with app-specific features offering marginal improvements.

Permission	Algorithm	Accuracy (words)	Accuracy (total)	Diff
Location	SVM	80.00%	81.74%	1.74%
	Maximum Entropy	81.97%	85.00%	3.03%
	C4.5	75.38%	79.57%	4.19%
Contacts	SVM	92.32%	93.94%	1.62%
	Maximum Entropy	93.57%	94.64%	1.07%
	C4.5	91.79%	92.86%	1.07%

DISCUSSION

In this section, we examine the limitations of our work and potential future improvements.

Code Obfuscation

In our experiments, we found that about 10% of apps contain obfuscated code, with much of it belonging to third-party libraries. Previous research [28] found that only about 2% of apps have obfuscated custom code. Our approach cannot handle these cases since we are unable to extract meaningful text-based features from the obfuscated identifier names.

More sophisticated program analysis techniques might mitigate this problem. For example, it may be possible to identify samples of code with similar purposes and structure. However, this is beyond the current scope of this paper.

Indirect Permission Use

We have also found that some apps use sensitive data through a level of indirection rather than directly accessing it. For example, the social networking app "`Skout`" has a package called "`com.skout.android.service`", containing services such as "`LocationService.java`" and "`ChatService.java`". In this design pattern, these services access sensitive data, with other parts of the app accessing these services. In this case, there was very little meaningful text information in the directory these services are located in, and our approach would simply fail.

One approach would be expanding the static analysis to look for this kind of design pattern. Another approach would be expanding the granularity of analysis from a directory to the entire app, and changing the classification from single-label classification to multi-label classification.

The Diversity of Developer Defined Features

Our approach is mainly based on text-based features. However, developers do not always use good identifier names, for example "v1" for a variable name. Developers also use abbreviations, for example using "loc" instead of "location". Our current splitting method does not work well for these cases.

One option is to manually label some known abbreviations. Another option is to use techniques such as approximate string matching [44] to infer abbreviated words.

Note that our approach also assumes that developers do not deliberately use misleading identifiers. If our approach becomes popular, a malicious developer could rename identifiers to confuse our classification. For example, a developer could rename identifiers to contain words such as "weather" or "temperature" to mislead how location data is used. Fortunately, we did not find any instances of this in our experimental data. It is also not immediately clear how to detect these kinds of cases either.

Incomplete List of Purposes

We have created a taxonomy of 10 purposes for the location permission and 10 purposes for the contacts permission, and used this taxonomy to manually label 1,020 instances. While our taxonomy is good enough for our purposes, it is possible that there are other purposes that we cannot find. Furthermore, depending on how purposes are used, our taxonomy might be too fine-grained or too coarse-grained.

However, we believe that our approach should generalize for new purposes and for other sensitive permission. For example, if there are more purposes for location data or contact list, we can simply add more training instances.

Inferring the Purpose of Third-Party Libraries

In this work, we only focused on the purpose of custom code, since past work [26, 27] has already looked at how to infer the purpose of third-party libraries (using a mapping from third-party library to purpose), and because many libraries are obfuscated. However, this past work only looked at the most popular third-party libraries, and there is a long-tail of third-party libraries that do not have a labeled purpose. Furthermore, some third-party libraries have multiple purposes.

It may be possible to apply our approach to infer the purpose of third-party libraries. However, we would need to expand the set of purposes (for example, location data would require "advertising" and "mobile payment" as a purpose) and rigorously evaluate the effectiveness of the approach.

CONCLUSION

In this paper, we have proposed a new technique to infer the purpose of sensitive information usage in custom written code. We extract different kinds of features from decompiled code, including both app-specific features and text-based features. These features are used to train a machine learning classifier. We evaluated our approach on two sensitive permissions, *location* and *contacts*. Experiments on 1,020 labeled instances show that our approach can successfully infer the purposes for about 85% of location permission uses and 94% of contacts permission uses. We also present a qualitative analysis on where our classifier works well and where it does not.

Acknowledgment

This work is supported by the High-Tech Research and Development Program (863) of China under Grant No.2015AA01A203, the National Natural Science Foundation of China under Grant No.61421091, 61103026. This work is based upon work supported by the National Science Foundation under Grant No.CNS-1228813.

REFERENCES

1. PrivacyGrade: Grading The Privacy of Smartphone Apps. `http://privacygrade.org/`.
2. Almuhimedi, H., Schaub, F., Sadeh, N., Adjerid, I., Acquisti, A., Gluck, J., Cranor, L. F., and Agarwal, Y. Your location has been shared 5,398 times!: A field study on mobile app privacy nudging. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)* (2015), 787–796.
3. Amini, S., Lin, J., Hong, J. I., Lindqvist, J., and Zhang, J. Mobile application evaluation using automation and crowdsourcing. In *Proceedings of the PETools* (2013).
4. Apktool: a tool for reverse engineering Android apk files. `https://code.google.com/p/android-apktool/`.
5. Wikipedia *App Store (iOS)*. `http://en.wikipedia.org/wiki/App_Store_%28iOS%29`.
6. Au, K. W. Y., Zhou, Y. F., Huang, Z., and Lie, D. Pscout: Analyzing the Android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)* (2012), 217–228.
7. Balebako, R., Jung, J., Lu, W., Cranor, L. F., and Nguyen, C. "little brothers watching you": Raising awareness of data leaks on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS '13)* (2013), 12:1–12:11.
8. Bartel, A., Klein, J., Le Traon, Y., and Monperrus, M. Automatically securing permission-based software by reducing the attack surface: An application to Android. In *the 27th IEEE/ACM Intl Conf on Automated Software Engineering (ASE '12)* (2012).
9. Bartel, A., Klein, J., Monperrus, M., and Le Traon, Y. Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android. *IEEE Transactions on Software Engineering (TSE)* (2014).
10. Wikipedia *C4.5 Algorithm*. `http://en.wikipedia.org/wiki/C4.5_algorithm`.
11. Chin, E., Felt, A. P., Sekar, V., and Wagner, D. Measuring user confidence in smartphone security and privacy. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12)* (2012).
12. Wikipedia *Cross-validation*. `http://en.wikipedia.org/wiki/Cross-validation_%28statistics%29#k-fold_cross-validation`.
13. dex2jar. `https://code.google.com/p/dex2jar/`.
14. Egelman, S., Felt, A. P., and Wagner, D. Choice architecture and smartphone privacy: Theres a price for that. In *Workshop on the Economics of Information Security (WEIS)* (2012).
15. Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security (SEC '11)* (2011).
16. Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *the 18th ACM Conference on Computer and Communications Security (CCS '11)* (2011), 627–638.
17. Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12)* (2012), 3:1–3:14.
18. Wikipedia *Google Play*. `http://en.wikipedia.org/wiki/Google_Play`.
19. Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)* (2014), 1025–1035.
20. Harbach, M., Hettig, M., Weber, S., and Smith, M. Using personal examples to improve risk communication for security and privacy decisions. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)* (2014).
21. Ismail, Q., Ahmed, T., Kapadia, A., and Reiter, M. Crowdsourced exploration of security configurations. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '15)* (2015).
22. JD-Core-Java. `http://jd.benow.ca/`.
23. Jing, Y., Ahn, G.-J., Zhao, Z., and Hu, H. Riskmon: Continuous and automated risk assessment of mobile applications. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy (CODASPY '14)* (2014), 99–110.
24. Jung, J., Han, S., and Wetherall, D. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '12)* (2012), 45–50.
25. Kelley, P. G., Cranor, L. F., and Sadeh, N. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13)* (2013), 3393–3402.
26. Lin, J., Amini, S., Hong, J. I., Sadeh, N., Lindqvist, J., and Zhang, J. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp '12)* (2012), 501–510.
27. Lin, J., Liu, B., Sadeh, N., and Hong, J. I. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Proceedings of the 2014 Symposium On Usable Privacy and Security (SOUPS '14)* (2014).

28. Linares-Vásquez, M., Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)* (2014), 242–251.

29. *Mallet:* machine learning for language toolkit. `http://mallet.cs.umass.edu/`.

30. Mancini, C., Thomas, K., Rogers, Y., Price, B. A., Jedrzejczyk, L., Bandara, A. K., Joinson, A. N., and Nuseibeh, B. From spaces to places: Emerging contexts in mobile privacy. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp '09)* (2009), 1–10.

31. Wikipedia *Maximum Entropy*. `http://en.wikipedia.org/wiki/Maximum_entropy`.

32. Evaluation methods in text categorization. `http://datamin.ubbcluj.ro/wiki/index.php/Evaluation_methods_in_text_categorization`.

33. Macro- and micro-averaged evaluation measures. `http://digitalcommons.library.tmc.edu/cgi/viewcontent.cgi?article=1026&context=uthshis_dissertations`.

34. Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security (SEC '13)* (2013), 527–542.

35. Permission Mappings. `http://pscout.csl.toronto.edu/`.

36. The porter stemming algorithm. `http://tartarus.org/martin/PorterStemmer/`.

37. Documented api calls mappings. `http://pscout.csl.toronto.edu/download.php?file=results/jellybean_publishedapimapping`.

38. Content provider (uri strings) with permissions. `http://pscout.csl.toronto.edu/download.php?file=results/jellybean_contentproviderpermission`.

39. Intents with permissions. `http://pscout.csl.toronto.edu/download.php?file=results/jellybean_intentpermissions`.

40. Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., and Chen, Z. 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), 1354–1365.

41. *Scikit-learn* machine learning in python. `http://scikit-learn.org/stable/index.html`.

42. Shih, F., Liccardi, I., and Weitzner, D. Privacy tipping points in smartphones privacy preferences. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)* (2015), 807–816.

43. Shklovski, I., Mainwaring, S. D., Skúladóttir, H. H., and Borgthorsson, H. Leakiness and creepiness in app space: Perceptions of privacy and mobile app use. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)* (2014), 2347–2356.

44. Wikipedia *Approximate String Matching*. `http://en.wikipedia.org/wiki/Approximate_string_matching`.

45. Supervised Learning. `http://en.wikipedia.org/wiki/Supervised_learning`.

46. Wikipedia *Support Vector Machine*. `http://en.wikipedia.org/wiki/Support_vector_machine`.

47. Toch, E., Cranshaw, J., Drielsma, P. H., Tsai, J. Y., Kelley, P. G., Springfield, J., Cranor, L., Hong, J., and Sadeh, N. Empirical models of privacy in location sharing. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp '10)* (2010), 129–138.

48. Wang, J., and Chen, Q. Aspg: Generating Android semantic permissions. In *Proceedings of the IEEE 17th International Conference on Computational Science and Engineering* (2014), 591–598.

49. English Wordlist. `http://www-personal.umich.edu/~jlawler/wordlist`.

50. Wu, L., Grace, M., Zhou, Y., Wu, C., and Jiang, X. The impact of vendor customizations on Android security. In *the 2013 ACM SIGSAC Conference on Computer Communications Security (CCS '13)* (2013), 623–634.

51. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., and Wang, X. S. Appintent: analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS '13)* (2013), 1043–1054.