Understanding the Purpose of Permission Use in Mobile Apps

HAOYU WANG, Beijing University of Posts and Telecommunications
YUANCHUN LI and YAO GUO, Peking University
YUVRAJ AGARWAL and JASON I. HONG, Carnegie Mellon University

Mobile apps frequently request access to sensitive data, such as location and contacts. Understanding the purpose of why sensitive data is accessed could help improve privacy as well as enable new kinds of access control. In this article, we propose a text mining based method to infer the purpose of sensitive data access by Android apps. The key idea we propose is to extract multiple features from app code and then use those features to train a machine learning classifier for purpose inference. We present the design, implementation, and evaluation of two complementary approaches to infer the purpose of permission use, first using purely static analysis, and then using primarily dynamic analysis. We also discuss the pros and cons of both approaches and the trade-offs involved.

CCS Concepts: • **Security and privacy** → **Mobile platform security**; **Privacy protections**; **Usability in security and privacy**; • **Information systems** → *Retrieval on mobile devices*; • **Human-centered computing** → *Mobile phones;*

Additional Key Words and Phrases: Permission, purpose, mobile applications, Android, privacy, access control

ACM Reference Format:
Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I. Hong. 2017. Understanding the purpose of permission use in mobile apps. ACM Trans. Inf. Syst. 35, 4, Article 43 (July 2017), 40 pages.
DOI: http://dx.doi.org/10.1145/3086677

1. INTRODUCTION

Mobile apps have seen widespread adoption, with over 2 million apps in both Google Play and the Apple App Store, and billions of downloads [AppStore 2016; GooglePlay 2016]. Mobile apps can make use of the numerous capabilities of a smartphone, which include a myriad of sensors (e.g., GPS, camera, and microphone) and a wealth of personal information (e.g., contact lists, emails, photos, and call logs).

This work was partly supported by the Funds for Creative Research Groups of China under Grant No. 61421061, the Beijing Training Project for the Leading Talents in S&T under Grant No. ljrc 201502, the National Natural Science Foundation of China under Grant No. 61421091, and the National High Technology Research and Development Program of China (863 Program) under Grant No. 2015AA017202. Jason Hong's work was supported in part by the Air Force Research Laboratory under agreement number FA8750-15-2-0281. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

Authors' addresses: H. Wang, Beijing Key Laboratory of Intelligent Telecommunication Software and Multimedia, School of Computer Science, Beijing University of Posts and Telecommunications; email: haoyuwang@bupt.edu.cn; Y. Li and Y. Guo (corresponding author), Key Laboratory on High-Confidence Software Technologies (MOE), School of Electronics Engineering and Computer Science, Peking University; emails: {liyuanchun, yaoguo}@pku.edu.cn; Y. Agarwal and J. I. Hong, School of Computer Science, Human Computer Interaction Institute, Carnegie Mellon University; emails: {yuvraj.agarwal, jasonh}@cs.cmu.edu.

DOI: http://dx.doi.org/10.1145/3086677

Mobile apps frequently request access to sensitive information, such as unique device ID, location data, and contact lists. Android currently requires developers to declare what permissions an app uses, but offers no formal mechanisms to specify the *purpose* of how the sensitive data will be used. While the latest Android releases have introduced permission strings to address this limitation, they are rarely used and only suggest a single purpose if they are used. Complicating this further, an app could use a permission for multiple purposes, such as using location permission for advertising, geotagging, and nearby searching. Mobile users have no way to know *how* and *why* a certain sensitive data item is used within an app, let alone controlling how the data should be used.

Knowing the purpose of a permission request can help with respect to privacy, for example, offering end-users more insights as to why an app is using a specific sensitive data. Prior work [Lin et al. 2012] showed that purpose information is important to assess people's privacy concerns. Properly informing users of the purpose of a resource access can ease users' privacy concerns to some extent. Besides, knowing a clear purpose of a request could also offer fine-grained access control, for example, disallowing the use of location data for geotagging while still allowing map searches.

Our specific focus is on developing better methods to infer the purpose of permission use. Prior work has investigated ways to bridge the semantic gap between users' expectations and app functionality. For example, WHYPER [Pandita et al. 2013] and AutoCog [Qu et al. 2014] apply natural language processing techniques to an app's description to infer permission use. CHABADA [Gorla et al. 2014] clusters apps by their descriptions to identify outliers in each cluster with respect to the Application Programming Interface (API) usage. RiskMon [Jing et al. 2014] 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 [Amini et al. 2013], 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. [2012, 2014], which introduced the idea of inferring the purpose of a permission by analyzing what third-party libraries an app uses. For example, if location data is only used by an advertising library, then it can be inferred that it is used for advertising. Lin et al. [2014] manually labeled 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. Their approach, however, is unable to detect purposes for sensitive data access within the app, particularly when there are multiple purposes (e.g., advertising, geotagging, etc.) for a single permission.

In this article, we propose a text mining based method to infer the purpose of a permission use for Android apps. A key insight underlying our work is that, unless an app has been completely obfuscated,[1] 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 possesses method or variable names such as "photo," "exif," or "tag," it is very likely that it uses location data for the purpose of "geotagging." We present two complementary approaches to determine the purpose of

[1] Note that if an app is fully obfuscated, we may not be able to infer the purpose of permission use. We detailedly analyzed the obfuscation rate in Android apps, the impact to our approach, and feasible approaches to deal with obfuscation in Section 6.1.

permission use based on text analysis: one using purely static analysis, the other using primarily dynamic analysis.

For static analysis we build upon our earlier work [Wang et al. 2015c], where we first decompile apps and search the decompiled code to determine where sensitive permissions are used. We have analyzed a large set of Android apps and from that data created a taxonomy of 10 purposes for location data and 10 purposes for contact list. The reason we chose contacts and location data is that past work has shown that users are particularly concerned about these two data items. 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 classifier to infer the purpose of permission uses.

However, relying on static analysis has some limitations. First, some apps use sensitive data through a level of indirection rather than directly accessing it. For example, the social networking app "`Skout`" has a helper package called "`com.skout.android.service`," containing services such as "`LocationService.java`" and "`ChatService.java`." In this design pattern, these helper services access sensitive data, with other parts of the app accessing these services instead. In this case, there is very little meaningful text information in the directory where these services are located, and static approach would simply fail to find enough context for purpose inference. Second, in many apps, third-party libraries request sensitive data by invoking methods in the app logic that provides access to resources, rather than accessing resources directly [Liu et al. 2015]. Furthermore, static analysis based approaches [Lin et al. 2012; Wang et al. 2015c] typically need to split apps into different components (e.g., libraries or packages) and label the purpose for each component. But specifying purpose at a component granularity is too coarse-grained as there may be multiple purposes of data use within each component.

To overcome the limitations of static analysis, we further introduce a dynamic approach to infer purpose at runtime. We use dynamic taint analysis at runtime to monitor privacy sensitive information flows, and infer the purpose of sensitive behavior based on dynamic call stack traces, which contain useful information on *how* (and *why*) the sensitive data is accessed and used. We extract meaningful key words from the methods and classes related to the call stack, and then use machine learning to infer the purpose of permission use. To infer the purposes accurately and address the multithreading programming patterns in Android, we propose a novel *thread-pairing* method to find the full stack trace at runtime.

We present the design, implementation, and evaluation of our static and dynamic approaches for inferring purposes in Android apps. *We first evaluate the effectiveness of text analysis techniques on decompiled code statically.* Our static analysis is focused on analyzing purposes for the custom code components of an app, excluding any included third-party libraries. We created a taxonomy for purposes on how apps use two sensitive permissions in custom code, namely, `ACCESS_FINE_LOCATION` (*location* for short) and `READ_CONTACTS` (*contacts* for short). We chose these two permissions as a proof of concept for our technique, in large part because past work has shown that users are particularly concerned about these two data items. For the static approach, we used this taxonomy to manually examine and label the behavior of 460 instances[2] using location (extracted from 305 apps), and 560 instances using contacts (extracted from 317 apps). We used this data to train a machine-learning classifier. Using 10-fold cross-validation,

[2] Here, an instance is defined as a directory of source code, thus a single app may yield more than one instance.

our experiments show that we can achieve about 85% accuracy in inferring the purpose of location use, and 94% for contact list use. *Then we introduce a dynamic analysis technique to overcome the limitations of static analysis.* For the dynamic approach, we try to infer the purpose of permission use in the entire app, including third-party libraries and custom code. We have implemented a prototype system that combined dynamic analysis and static analysis on Android, and we have evaluated the effectiveness of our system by testing it on 830 popular Android apps. Our experimental results show that we are able to successfully infer the purpose of over 90% of the sensitive data uses.

This article 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 the purposes for custom-written code (as opposed to third-party libraries or app descriptions).

—*We present the design, implementation, and evaluation of two complementary approaches to infer the purpose of permission use*, one using purely static analysis, the other using primarily dynamic analysis. We also created a taxonomy for purposes regarding how apps use location and contacts permissions. We show that both approaches are able to identify the purposes for 90% of the sensitive data uses on average.

—*We discuss the pros and cons of both the static approach and the dynamic approach, as well as the trade-offs involved.* Since the static approach has good code coverage and scalability, it is feasible to deploy it on the app market to identify sensitive behaviors of mobile apps a priori, and help improve user awareness about which permissions are used by an app and why. Our dynamic analysis is finer-grained and improves accuracy for purpose inference. It is therefore more suitable to deploy the dynamic approach on real users' phones and help them enforce privacy.

2. BACKGROUND AND RELATED WORK

2.1. Background

2.1.1. The Android Permission Mechanism. Android uses a permission model to govern an app's access to resources. Prior to Android Marshmallow (version 6.0), all permissions were declared by developers in a manifest file, and end-users were required to accept all of them at install time. Android Marshmallow introduced runtime permission control for several "dangerous" permissions such as location or contact list, allowing users to allow (or deny) access on first use. Furthermore, these permissions can be modified later if the user feels uncomfortable on granting the app access to a certain resource all the time. However, despite this additional control over permissions granted to individual apps, Android still lacks the capability to let users both understand and choose the *purpose* for which each permission is granted to an app. Once a user grants the access to an app, the requested data can be used for any purpose.

2.1.2. The Purpose of Permission Use. In this article, the *purpose* of a permission refers to *the reason for accessing a sensitive data item*, that is, why an app needs access to a specific sensitive data. For example, for an app that uses location data for turn-by-turn navigation and for advertising, one might say that this app uses location data for "navigation" and for "ads."

Prior work has shown that static analysis of apps can help identify libraries that use sensitive permissions and infer its purpose. Lin et al. [2012, 2014] manually categorized around 400 popular third-party libraries based on their functionality, and then used these categories to label the purposes of permissions used in each library. The libraries are categorized into nine different purposes, as shown in Table I. Note that we added

Table I. A Taxonomy of the Purposes of Permission Uses. Third-Party Libraries are Categorized into 10 Different Purposes [Lin et al. 2012]. We Manually Analyzed a Large Set of Android Apps and Created a Taxonomy of the Purposes of *Location* Permission Uses and the Purposes of *Contacts* Permission Uses in Custom Code

Type	Permission	Purpose
The purpose of permission use in third-party libs [Lin et al. 2012]	all permissions	`advertising`, `analytics`, `social networking`, `utilities`, `development aid`, `social games`, `secondary market`, `payment`, `game engine`, *`maps`*
The purpose of permission use in custom code	location	`search nearby places`, `location-based customization`, `transportation information`, `recording`, `map and navigation`, `geosocial networking`, `geotagging`, `location spoofing`, `alert and remind`, and `location-based game`
	contacts	`backup and synchronization`, `contact management`, `blacklist`, `call and SMS`, `contact-based customization`, `email`, `find friends`, `record`, `fake calls and SMS`, `remind`

Table II. Our Set of Purposes for Location Permission in Custom Code, and the Number of Unique Packages in Our Dataset that have that Purpose

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

a new category called "map library,"[3] which includes Software Development Tookits (SDKs) such as `osmdroid`.

For the purpose of permission use in custom code, we manually analyzed a large set of Android apps and created a taxonomy of the purposes of *location* permission use and the purposes of *contacts* permission use, as shown in Table I. The description of each purpose is detailedly explained in Tables II and III.

2.2. Related Work

2.2.1. The Gap Between User Expectations and App Behaviors. Past studies [Felt et al. 2012; Chin et al. 2012; Egelman et al. 2012] have shown that mobile users have a poor

[3] Note that purpose "maps" refers to the purpose of location data used in third-party map libraries, while the purpose "map and navigation" refers to the purpose of location data used in custom code for driving route planning and navigation.

Table III. Our Set of Purposes for Contacts Permission in Custom Code, and the Number of Unique Packages in Our Dataset that has that Purpose

Purpose	Description	#Instances
Backup and Synchronization	Backup contacts to the server, restore and sync contacts	61
Contact Management	Remove invalid contacts, delete/merge duplicate contacts	30
Blacklist	Block unwanted calls and SMS	52
Call and SMS	Make VoIP/Wifi calls using Internet, send text message	54
Contact-based Customization	Add contacts to a custom dictionary for input methods, change ringtone and background based on contacts	51
Email	Send email to contacts	78
Find friends	Add friends from contacts, find friends who use the app in contact list	46
Record	Call Recorder, call log and history	93
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
Remind	Missed call notification, remind you to call someone	46

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 [Felt et al. 2012; Shklovski et al. 2014]. Several approaches [Almuhimedi et al. 2015; Harbach et al. 2014; Kelley et al. 2013] 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 [Balebako et al. 2013; Jung et al. 2012] suggested that there is a semantic gap between users' expectations and app behaviors. Recent research 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 [Pandita et al. 2013], AutoCog [Qu et al. 2014], and ACODE [Watanabe et al. 2015] propose to 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. However, the results suggest that, for more than 90% of apps, it is impossible to understand why permissions are used based solely on app descriptions. ASPG [Wang and Chen 2014] has proposed generating semantic permissions using NLP techniques on app descriptions. 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 [Gorla et al. 2014] 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 that use APIs that are uncommon for that cluster. All of these approaches have attempted to infer permission use or semantic information from app descriptions, and bridge the gap between app descriptions and functionalities.

Ismail et al. [2015] leveraged crowdsourcing to find the minimal set of permissions to preserve the usability of an app for diverse users. RiskMon [Jing et al. 2014] 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 [Amini et al. 2013], a tool that combines crowdsourcing and dynamic analysis to help users understand and flag unusual behaviors of apps. AppIntent [Yang et al. 2013] uses symbolic execution to infer whether a transmission of sensitive data is by user intention or not. Past research [Shih et al. 2015; Mancini et al. 2009; Toch et al. 2010] has also attempted to measure users' privacy preferences in different contexts. For example, Shih et al. [2015] 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.

2.2.2. Fine-Grained Privacy Enforcement. Mobile privacy is a growing concern, while many research works have proposed to enforce privacy protection. One line of work is fine-grained controls to prevent access to sensitive information, including OS-level protection such as Kirin [Enck et al. 2009], Saint [Ongtang et al. 2009], APEX [Nauman et al. 2010], ProtectMyPrivacy [Agarwal and Hall 2013], FlaskDroid [Bugiel et al. 2013], ASF [Backes et al. 2014] and ASM [Heuser et al. 2014], and app-level protection through instrumentation such as Aurasium [Xu et al. 2012], AppGuard [Backes et al. 2013], I-arm-droid [Davis et al. 2012], RetroSkeleton [Davis and Chen 2013]. These approaches only prevent information from being accessed, while they typically do not consider how the sensitive information is used in the app.

Another line of work has extended the system to track information flows. TISSA [Zhou et al. 2011], MockDroid [Beresford et al. 2011], and AppFence [Hornyack et al. 2011] replace sensitive information with fake data. CleanOS [Tang et al. 2012] modifies TaintDroid to enable secure deletion of information from application memory. Kynoid [Schreckling et al. 2013] extends TaintDroid with user-defined security policies such as restrictions on destinations IP address to which data is released. BayesDroid [Tripp and Rubin 2014] is proposed for quantitative information flow analysis, which is to measure the amount of privacy information that can be inferred from the leaked data. FlowDroid [Arzt et al. 2014], DroidSafe [Gordon et al. 2015], and DroidInfer [Huang et al. 2015] use static information flow analysis to detect privacy leakage.

Another area of related work is focused on privilege separation of apps and ad libraries. Ad libraries share the same permissions with the host app, which can potentially lead to privacy issues. AdSplit [Shekhar et al. 2012] extends Android to allow an app and its Ad libraries to run as separated processes with different user IDs. AdDroid [Pearce et al. 2012] introduces new APIs and permissions for Ad libraries, which enables it to separate privileged advertising functionality from the host app. Roesner and Kohno [2013] propose to allow Android to permit ad libraries to embed User Interface (UI) elements in the main logic without exposing data or privileges of the main app. PEDAL [Liu et al. 2015] uses a machine-learning approach to identify Ad libraries first, then rewrites the resource access and resource sharing functions to enforce access control for Ad libraries.

These past works could detect privacy leaks or help enforce privacy, but do not investigate why an app is using sensitive data.

2.2.3. Determining the Purpose of Permission Uses. Understanding the purpose of why sensitive data is used could help improve privacy as well as enable new kinds of access control. Lin et al. [2012, 2014] 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.

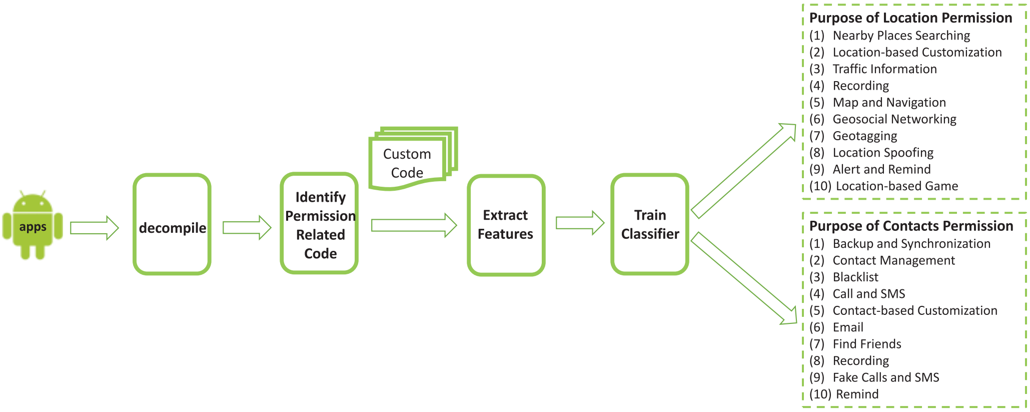


Fig. 1. The overall architecture of the static analysis approach. We first decompile each app and filter out third-party libraries using a list of the most popular libraries. We then use static analysis to identify where permission-related code is located. We extract several kinds of features from this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

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 [PrivacyGrade 2015; Wang et al. 2017] 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. We focus on inferring the purpose for two sensitive permissions: `location` and `contacts`. We chose these two permissions as a proof of concept for our technique, and believe that our approach should generalize to other permissions. Based on our analysis of more than 7,000 apps, we created a taxonomy of the purpose of location permission use and the purpose of contacts permission use, as shown earlier in Table I.

We present the design, implementation, and evaluation of two complementary approaches to infer the purpose of permission use, one using purely static analysis, the other using primarily dynamic analysis combined with static analysis.

3. INFERRING THE PURPOSE USING STATIC ANALYSIS

3.1. Overview

As shown in Figure 1, we first use static analysis to identify the corresponding custom code that uses the 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 the 10 different purposes for `location` or one of the 10 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, and partly because we found that many third-party libraries were obfuscated, which makes static analysis and text mining more difficult.

3.2. Decompiling Apps

For each app, we first decompile it from DEX (Dalvik Executable) into intermediate *Smali* code using Apktool [2016]. Smali is a kind of register-based language, and one Smali file corresponds to exactly one corresponding Java file. We use Smali because we found that it is easier to identify permission-related code based on this format.

We then decompile each app to *Java* using dex2jar [Dex2jar 2016] and JD-Core-Java [JD-Core-Java 2016]. We use the decompiled Java source code to extract features. Previous research [Enck et al. 2011] 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 during our static analysis experiments. In Section 6.1, we will measure the code obfuscation rate in current Android apps, measure the effectiveness of our approach, and explore feasible ways to deal with code obfuscation.

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 third-party libraries built by past work [Lin et al. 2012] to remove libraries; we found that it works reasonably well in practice, in large part due to a long tail distribution of the libraries used in Android apps.

3.3. 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 [PermissionMappings 2015] provided by PScout [Au et al. 2012] to determine which permissions are actually used in the code and where they are used. More specifically, we created a lightweight analyzer for searching sensitive API invocations, Intents, and Content Providers in the Smali code. For example, if we find the Android API string "`Landroid/location/LocationManager;->getLastKnownLocation`" in the code, we know it uses the `location` permission. Since the 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 geotagging, nearby searching, and advertisement, but a coarse-grained approach might not find all of these purposes. Another option is applying 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 static approach, we decided to use all of the classes in the same `directory` as the level of granularity. In Java, a directory (or file folder) very often maps directly to a single package, although 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. Here 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.

Table IV. The Features Used in the Classification Model

Type	Feature	Feature Description	Representation	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		

3.4. 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 IV. App-specific features are based on app behaviors and code functionality, while text-based features rely on meaningful identifier names as given by developers.

3.4.1. 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 normalize 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 [Au et al. 2012]), 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 [PScout API 2015], 97 kinds of Intents associated with permissions [PScout Intent 2015], and 78 kinds of Content Provider URI Strings associated with permissions [PScout ContentProvider 2015]. 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" purposes. Thus, for instances that use such APIs, it is quite possible that it uses contacts for "email" purposes.

We use a list of 680 documented APIs that correlate to 51 permissions provided by Pscout [PScout API 2015], 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 [PScout Intent 2015] and 78 Content Provider URI strings [PScout ContentProvider 2015]. 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 occurrences of the corresponding Intent or Content Provider.

3.4.2. 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, although 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 words "find" and "restaurant." Thus, we need to split identifiers appropriately to extract relevant words. Third, not all words are equally useful, and so we need to 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*, although underscores are sometimes used. For identifiers with explicit delimiters, we use their construction patterns to split them into subwords. The identifier patterns we used are as listed as follows:

$camelcase(1) : AbcDef \rightarrow Abc, Def$
$camelcase(2) : AbcDEF \rightarrow Abc, DEF$
$camelcase(3) : abcDef \rightarrow abc, Def$
$camelcase(4) : abcDEF \rightarrow abc, DEF$
$camelcase(5) : ABCDef \rightarrow ABC, Def$
$underscore : ABC\_def \rightarrow ABC, def$

ALGORITHM 1: Dictionary-Based Identifier Splitting Algorithm

```
Input: $identifierI$ and $wordlist$
Output: a list of splitted $keywords$
initial $keywords$ = NULL
$subword \leftarrow FindLongestWord(I, wordlist)$
while $subword \neq NULL$ and $len(I) > 0$ do
    keywords.add(subword)
    if $len(I) = len(subword)$ then
        break
    end if
    $I \leftarrow identifier.substring(len(subword), len(I))$
    $subword \leftarrow FindLongestWord(I, wordlist)$
end while
```

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 [WordList 2015]. 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 [Porter 2015] to stem 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.

3.5. 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 [2016], Maximum Entropy [2016], and C4.5 Decision Tree [C4.5 2016]. The implementation of SVM is based on the python scikit-learn [SciKit 2016] package. We use a Support Vector Machine (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 [2016]. We then compare different classifiers using various metrics.

3.6. Evaluation

3.6.1. Dataset. We downloaded 7,923 apps from Google Play, 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 approach, which requires labeled instances. We focus on apps that use 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 tried to manually label at least 50 instances for each purpose. For the location permission, we had more than 3,000 instances in our dataset, so we stopped once we got more than 50 examples for a given purpose. As shown in Table I, 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 the contacts permission, we found fewer than 800 instances in our dataset, so we manually checked and labeled the purposes for all these instances (which is why the number of instances in Table II are not as uniform as those in Table I).

Purpose Labeling Process. 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. It is true that for several instances, due to code obfuscation[4] or indirect permission use, we cannot spell its purpose in our previous static analysis and we omit these instances when we label the ground truth. But for many instances, we could infer its purpose accurately. For example, in one case, we found custom code using location data, including method and variable names containing words such as “temperature” and “wind,” which we labeled as “location-based customization.” As another example, we found an 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 identify 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 observation matches previously reported results [Qu et al. 2014], which found that for more than 90% of apps, users could not understand why permissions are used based solely on descriptions. This 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 Tables II and III.

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 feel that our dataset is good enough as an initial demonstration of our idea. We will offer more details on this issue in Sections 4 and 5.

3.6.2. Evaluation Method. We used 10-fold cross-validation [Cross-Validation 2016] 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

[4] We will detailedly analyze the impact of code obfuscation in Section 6.

Table V. The Results of Inferring the Purpose of Location Uses

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

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 (*TPs*), False Positives (*FPs*), True Negatives (*TNs*), and False Negatives (*FNs*). We also present our results in terms of *precision*, *recall*, and *f-measure*. Precision is defined as the ratio of the number of TPs 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 *microaveraged* and *macroaveraged* metrics to measure the precision and recall. For microaveraged metrics, we first sum up the *TPs*, *FPs*, and *FNs* for all the classes, and then calculate precision and recall using these sums. In contrast, macroaveraged scores are calculated by first calculating precision and recall for each class and then taking the average of them. Microaveraging is an average over instances, and so classes that have many instances are given more importance. In contrast, macroaveraging gives equal weight to every class. We calculate microaveraged precision, microaveraged recall, macroaveraged precision, and macroaveraged recall as follows, where *c* is the number of different classes.

$$MicroAvg_{Precision} = \frac{\sum_{i=1}^{c} TPi}{\sum_{i=1}^{c} TPi + \sum_{i=1}^{c} FPi}, \tag{1}$$

$$MicroAvg_{Recall} = \frac{\sum_{i=1}^{c} TPi}{\sum_{i=1}^{c} TPi + \sum_{i=1}^{c} FNi}, \tag{2}$$

$$MacroAvg_{Precision} = \frac{\sum_{i=1}^{c} Precision_i}{c}, \tag{3}$$

$$MacroAvg_{Recall} = \frac{\sum_{i=1}^{c} Recall_i}{c}. \tag{4}$$

Note that both microaveraged precision and microaveraged recall are equal to the *accuracy* of the classifier in our experiment. Thus, we only list the *accuracy* and *macroaveraged metrics* in Tables V and VIII.

3.6.3. Results of Inferring Location Purposes. Table V shows our results in classifying the purpose of *location*. 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 VI 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 VII shows more details about misclassifications. The category "search

Table VI. 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.

Table VII. The Confusion Matrix of Inferring the Purpose of Location Permission Use (Maximum Entropy). The Purpose Number (e.g., L1, L2, etc) Corresponds to that Listed in Table VI. 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 VIII. The Results of Inferring the Purpose of Contacts Permission Uses

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

nearby places" has the most false positives (see column L1, 14 of 56 classified instances), and four misclassified instances belong to the "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."

3.6.4. Results of Inferring Contacts Purposes. Table VIII shows our results for inferring the purpose of contacts. All three classification algorithms have achieved better than 90% accuracy, with the Maximum Entropy classifier still performing the best at 94.64%.

Table IX presents the 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, the "contact management" category is not as good, with both

Table IX. 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 X. The Confusion Matrix of Inferring the Purpose of Contacts Permission Use (Maximum Entropy). The Purpose Number (e.g., C1, C2, etc.) Corresponds to that Listed in Table IX. 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

precision and recall under 85%. Table X 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.

3.6.5. 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 roles in the classification: *distinctive features* and *the number of features*.

Categories with high precision and recall tend to have distinctive features. For example, instances in "location-based customization" have words like "weather," "temperature," and "wind," which are very rare in other categories. In contrast, misclassified instances have more generic words. For example, the labeled instance "`com.etech.placesnearme`" uses location information to search nearby places, and its top key words were "local," "search," "place," "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,

Table XI. 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)	Difference
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%

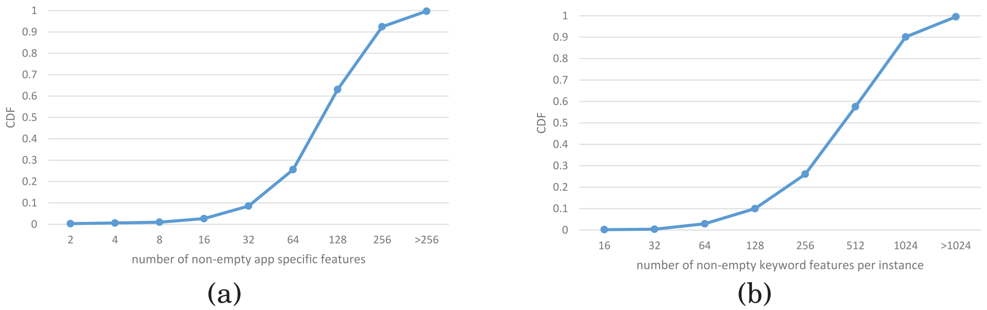


Fig. 2. The distribution of the number of nonempty (a) app-specific features and (b) text-based features per instance.

"`com.flashlight.lite.gps.passive`" uses location information for "recording." However, it only has 19 kinds of word features and six 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.

3.6.6. Feature Comparison. We are also curious how well text-based features alone are able to perform in the process, since that is one of the key novel aspects 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 XI.

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 that text-based features alone perform very well, while app-specific features play a supporting role.

Figure 2 offers one possible explanation. It shows the number of nonempty app-specific features and nonempty 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 are more dominant in the classifier. The number of text-based features for each instance is about four 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 1,024.

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.

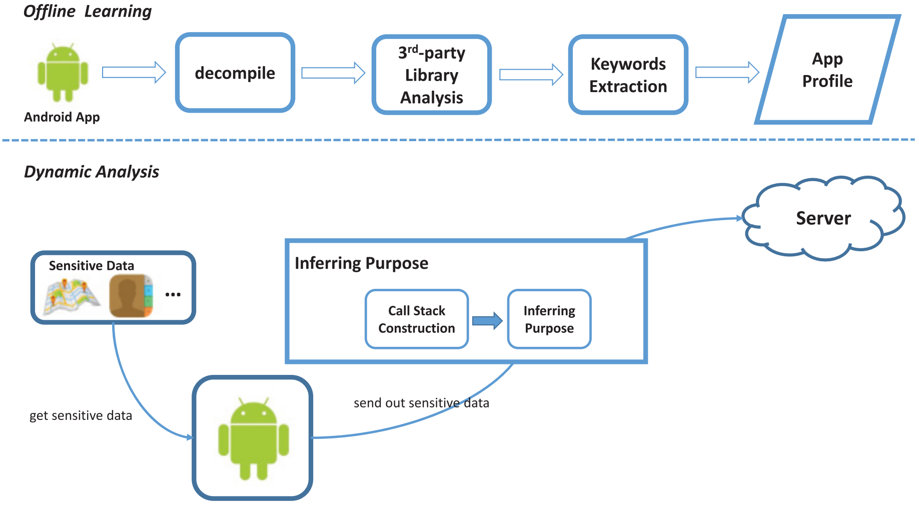
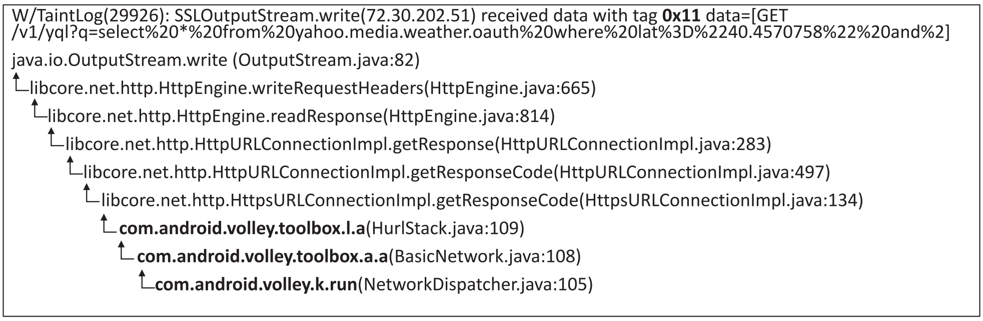


Fig. 3. Overall architecture of our dynamic analysis approach for inferring the purpose of a permission. At runtime, our system uses dynamic taint analysis to track sensitive data propagation. Once an app is about to leak the sensitive data, our system will construct the call stack and analyze its purpose using a library-based method in combination with text-based techniques with the aid of the app profile. We use offline learning (static analysis) to improve the accuracy of purpose inference by statically analyzing each app beforehand to build its profile.

4. INFERRING PURPOSES AT RUNTIME

Relying on static analysis to infer the purpose has several limitations. First, in many cases, the sensitive data invocation is *indirect*. For example, many apps use a particular design pattern where one part of the app periodically accesses and caches the sensitive data, while other parts of the app accesses that data asynchronously. Second, in many apps, third-party libraries request sensitive data by invoking methods in the app logic that provides access to resources, rather than accessing resources directly [Liu et al. 2015]. Furthermore, specifying purpose at a package granularity is too coarse-grained as there may be multiple purposes of data use in each package.

To overcome these limitations of static analysis, we introduce a call stack based method to infer the purpose of sensitive permission uses at runtime. By analyzing the call stack, we can learn which classes and methods access the sensitive data and how that data is used. In combination, these techniques offer a hint as to why sensitive data is being used. The overall architecture of our dynamic analysis approach for inferring the purpose of a permission is shown in Figure 3. We use dynamic taint analysis to track the flow of sensitive data. Here, we take advantage of a modified version of TaintDroid [Enck et al. 2010]. We analyze the call stack at taint sink points (e.g., network interface) to infer the purpose of privacy leakage. We choose to infer purpose at the sink point because using sensitive data at the source and intermediate points does not always lead to privacy leakage (used within the app client). Besides, because we build the full call stack traces, we could capture the information of acquired resources and how the information is used at sink point. On one hand, the call stack directly reflects how the resource is used (the sink); on the other hand, we are able to know which resource is accessed (the source) using dynamic taint tracking. For example, at the sink point, we can check the taint tag of sinked data to know where the data come from, and how the sensitive data is used within the app and what the data is used for using the call stack traces.



```
W/TaintLog(29926): SSLOutputStream.write(72.30.202.51) received data with tag 0x11 data=[GET
/v1/yql?q=select%20*%20from%20yahoo.media.weather.oauth%20where%20lat%3D%2240.4570758%22%20and%2]

java.io.OutputStream.write (OutputStream.java:82)
↑
└libcore.net.http.HttpEngine.writeRequestHeaders(HttpEngine.java:665)
  ↑
  └libcore.net.http.HttpEngine.readResponse(HttpEngine.java:814)
    ↑
    └libcore.net.http.HttpURLConnectionImpl.getResponse(HttpURLConnectionImpl.java:283)
      ↑
      └libcore.net.http.HttpURLConnectionImpl.getResponseCode(HttpURLConnectionImpl.java:497)
        ↑
        └libcore.net.http.HttpsURLConnectionImpl.getResponseCode(HttpsURLConnectionImpl.java:134)
          ↑
          └com.android.volley.toolbox.l.a(HurlStack.java:109)
            ↑
            └com.android.volley.toolbox.a.a(BasicNetwork.java:108)
              ↑
              └com.android.volley.k.run(NetworkDispatcher.java:105)
```

Fig. 4. An example call stack from the Yahoo Weather app showing the challenge of stack traces with multithreading. The app tried to send location data (tag 0x11) to a remote server. However, due to a common design pattern, when we get the call stack at a taint sink, we only get it from the current child thread. As a result, a great deal of potentially useful information has been lost.

More specifically, we examine the call stack for well-known libraries and use machine-learning techniques on key words in the call stack to infer the purpose. Because the call stack often does not contain enough information by itself, and since package names are sometimes obfuscated, we also introduce an *offline learning* step to statically analyze each app beforehand to build the app profile. This profile includes the third-party libraries used in the app and key words extracted from each class. The purpose can then be inferred based on all this information dynamically. Thus, our approach combines both dynamic analysis and static analysis.

4.1. Constructing the Call Stack

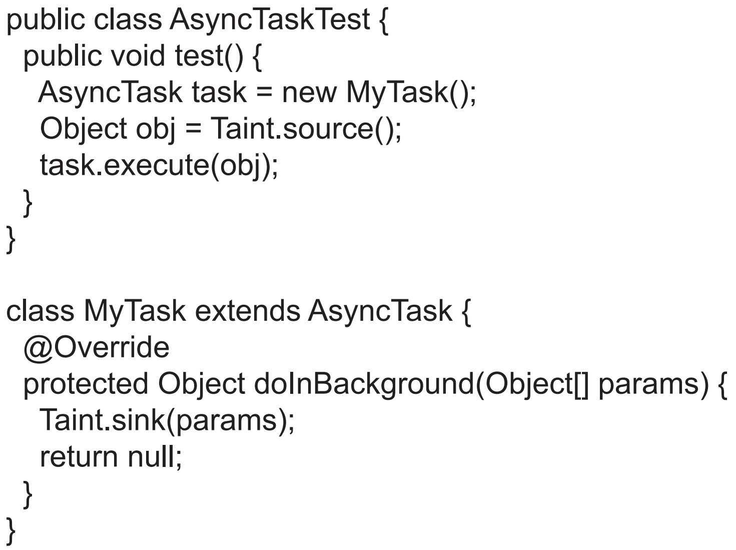
Several Java APIs (e.g., `printCallStack()`) can be used to get stack traces of the current thread in Android. However, Android apps are often programmed as multithreaded, making it difficult to infer the purpose using just the call stack of the current thread. For example, one common design pattern in Android apps is to request sensitive data (such as getting location) in the parent thread, and then spawn another thread to send sensitive data to a remote server. One such instance is the *Yahoo Weather*[5] app. When we get to the sink point (see Figure 4), we can only get the call stack of the child thread, which only shows rather ordinary network behaviors using the *volley* HTTP library.

Thus, to improve dynamic runtime analysis, we need to retrieve not only the call stack trace of the current thread, but also other threads related to the current thread. There are three common design patterns for how developers use threads in Android [MultipleThreads 2016]:

—**Pattern 1: Using Java thread APIs**. Java provides a set of low-level APIs to allow a program to create threads and start them immediately. More specifically, the parent thread first creates a new `Thread` instance, implementing a callback function such as `run()`. It can then start the child thread by invoking method `start()`.

—**Pattern 2: Android platform-specific APIs based on `ThreadPool`**. Android manages threads with a thread pool, which is implemented in the class `ThreadPoolExecutor`. Most high-level Android thread APIs such as `AsyncTask` and `ScheduledThreadPoolExecutor` are implemented based on ThreadPool. ThreadPool manages a set of threads and a queue of tasks, and dispatches tasks one by one when there are available threads. These APIs are good encapsulations of the Java `Thread` class.

[5] com.yahoo.mobile.client.android.weather.



```
public class AsyncTaskTest {
  public void test() {
    AsyncTask task = new MyTask();
    Object obj = Taint.source();
    task.execute(obj);
  }
}

class MyTask extends AsyncTask {
  @Override
  protected Object doInBackground(Object[] params) {
    Taint.sink(params);
    return null;
  }
}
```

Fig. 5. Usage example of `AsyncTask`. Two methods (`execute` and `doInBackground`) work together to accomplish asynchronous tasks.

—**Pattern 3: Looper-based multithread APIs in Android**. Looper [2016] is a Java class within Android that, together with the `Handler` class, can be used to process UI events such as button clicks. In Android, the main thread (the UI thread) keeps looping in the background and waits for messages from other threads. Once a message is received, the main thread starts to process the message. The `Handler` class and `Message` class, which are typically used in updating UI from non-UI threads, are based on Android Looper.

4.1.1. Identifying the Full Call Stack Trace. There are often some shared objects between the current thread and its related threads, which can be used to identify connections between threads and uncover related stack traces. To identify the *thread bridges*, we use a heuristic thread-pairing approach at runtime.

For example, as shown in Figure 5, consider the class `AsyncTask` with two methods (`execute` and `doInBackground`) that work together to accomplish asynchronous tasks, while they share the same `AsyncTask` instance object. To use the `AsyncTask` API, the developer should implement the `doInBackground` callback and call `execute` to start an asynchronous task. The `execute` method is called in the parent (caller) thread, which will create a child (callee) thread and pass arguments to it while `doInBackground` is then called from the callee thread.

When we tried to get the call stack trace at the taint sink (in method `doInBackground`), we can only get the call stack trace of the child thread, which missed potentially useful information in the parent thread.

However, the `AsyncTask` instance shared between the two threads can help us find the connection between them. The child thread knows the task it is executing (by referring to `this` object in `doInBackground`), which is the same `task` object used by the parent thread to start the child thread. By comparing the objects shared between threads, we are able to find the corresponding parent thread.

The other kinds of multithread programming patterns are similar to this `AsyncTask` example. Thus, we introduce a thread-pairing approach to identify the thread bridges (shared objects) between threads:

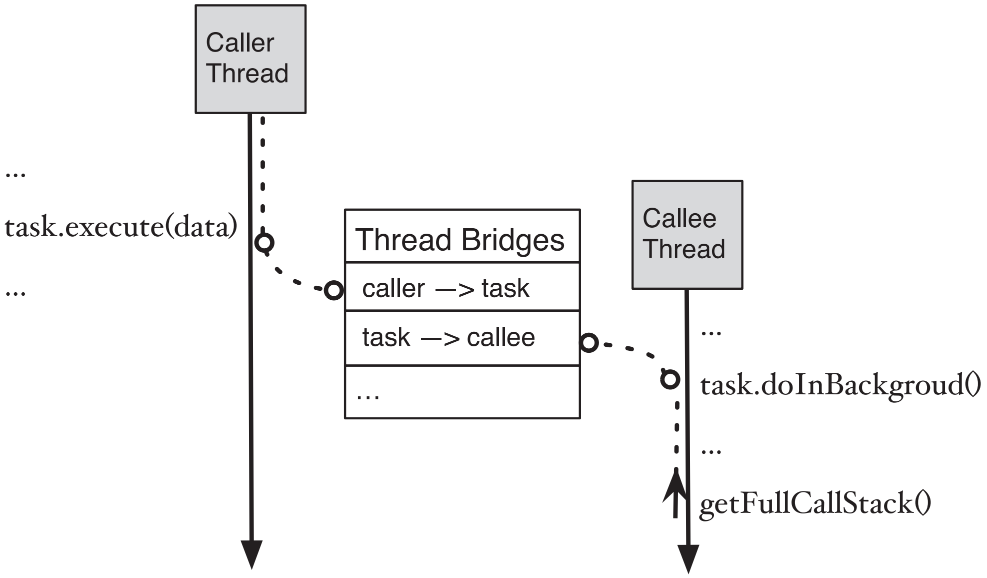


Fig. 6. A bridge-building example in the `AsyncTask` API. The two threads share the same AsyncTask instance object, which can be used to find the connection between them.

—For threads using Java thread APIs, the caller and callee threads share the same child `Thread` instance.
—The threads using the `ThreadPool` share the same task instance with their children threads.
—The caller threads using `Handler` share the identical `Message` instance with the main thread.

To implement multithread stack trace tracking in Android, we modified *Dalvik* to maintain a bridge-thread mapping during runtime. First, we located the key APIs of the three multithread programming patterns in Android source code and identified the shared instances (bridges). Then we instrumented these APIs to connect related threads. For example, in the `AsyncTask` shown in Figure 6, we built a caller-to-instance bridge after the `execute` method and an instance-to-callee bridge before the `doInBackground` method.

Note that our system takes a snapshot of the caller stack when the caller thread invokes a method to start a new thread. When getting the full call stack, we first get the call stack of the current thread with the `getStackTrace()` API, look up the bridges to find the parent threads, then we read the call stack snapshots of the parent threads from memory, and finally we concatenate the stacks together to form a full call stack. Because the caller stack we used is a snapshot of when the caller thread tries to start the callee thread, we are fully convinced that the caller stack is deterministic in our implementation. We discuss the implementation details in Section 4.3.

4.2. Inferring Purpose Based on Call Stack

Based on the call stack, we use two heuristics to infer the purpose. We first analyze the call stack traces to see whether the sensitive data is used by well-known third-party libraries (e.g., advertising libraries) based on a previously labeled list of popular libraries [Lin et al. 2012]. If a well-known library is not found, then we use a text-based machine-learning method, which demonstrated to be effective in our static approach.

We extract meaningful key words from the methods and classes that related to the call stack, and calculate the TF-IDF as features, and then feed it to a machine-learning classifier to learn the purpose.

4.2.1. Challenges. Even after linking stack traces from multiple threads together, there are still two substantial challenges in inferring the purpose:

—Prior research [Liu et al. 2015] shows that many third-party libraries use obfuscation, making it hard to identify third-party libraries using package names alone, let alone knowing the purposes.
—While call stacks contain package names, class names, and method names, this information is sometimes still not enough for inferring purposes.

4.2.2. Extracting App Profile. To address these two challenges, we generate an *app profile* beforehand using static analysis, which is then used to help infer purposes at runtime.

We use static analysis in two ways. First, we identify third-party libraries that may be obfuscated. To do this, we use a clustering-based approach [Ma et al. 2016; Wang et al. 2015a] to identify third-party libraries in the app based on API features rather than comparing package names. Then we use a categorization of about 400 popular third-party libraries labeled previously [Lin et al. 2012, 2014] to label the purpose of sensitive data used by these third-party libraries. Note that the categorization is somewhat outdated, thus we added some new libraries, and added a new category called "map library" that includes SDKs such as `osmdroid`.

Second, we extract additional identifiers such as field names and method names in the same class, which can also offer some hints to infer the purpose. We process the decompiled code and extract meaningful key words from identifier names for each class. Based on the results, we can extend the key words extracted from call stack. The features we use contain not only the key words that appear in the call stack, but also the key words extracted from various kinds of identifier names (field names, class names, method names) from classes used in the call stack. To extract keywords for each class, we apply *identifier splitting* as introduced in Section 3.

4.2.3. Inferring Purpose at Runtime. Based on the call stack traces and app profile, our dynamic purpose inferring algorithm is comprised of the following steps:

—We first check for sensitive dataflows through third-party libraries using the previously built app profiles. If this sensitive data is used by a known third-party library, we label its purpose directly. Otherwise, the sensitive data is used by the custom app code.
—Based on the call stack and app profile, we identify the classes used in the call stack, and combine the key words used in them. Then we calculate the TF-IDF vector as features. IDF is calculated based on a corpus of 2,000 apps.
—Finally, we use a pretrained SVM model to infer the purpose. The SVM classifier is trained offline with 460 instances labeled in our static approach (Section 3). We implement the SVM classifier in the Android *libcore*, and the classifier runs entirely on the Android device.

Note that, to improve the performance of purpose inferring at runtime, we calculate the TF-IDF for each class when creating an app profile. At runtime, when we need to extract features for a call stack, we first find used classes in the call stack and then calculate the new feature vector based on the TF-IDF vectors of related classes. Let $f_c(word_i)$ be the TF-IDF result for $word_i$ in class c, $Count_c\,(word_i)$ is the term frequency of $word_i$ in class c, and $IDF(word_i)$ is the inverse document frequency of $word_i$. If the call stack contains two related class $c1$ and $c2$, the TF-IDF result for $word_i$ in the call stack can be calculated as

$$
\begin{aligned}
f_{c1}(word_i) &= \frac{Count_{c1}(word_i)}{Total_{c1}} \times IDF(word_i),\\
f_{c2}(word_i) &= \frac{Count_{c2}(word_i)}{Total_{c2}} \times IDF(word_i),\\
f_{call-stack}(word_i) &= \frac{Total_{c1} \times f_{c1}(word_i) + Total_{c2} \times f_{c2}(word_i)}{Total_{c1} + Total_{c2}}.
\end{aligned}
$$

4.2.4. Optimization with Purpose Caching. We discovered significant repetition of several call stack traces, meaning that the app was trying to send the same sensitive data to a remote server multiple times. In most apps, the number of *unique* sensitive call stack traces is small (less than 10), providing an opportunity to optimize the runtime performance.

To improve the runtime performance, we introduce *purpose caching*, which involves caching and reusing previous inferences of the exact same call stack. To enable efficient comparison, we use a lightweight format to represent the call stack trace, which is comprised of a *quad* including the *destination IP address*, *sensitive data type*, the *length of the call stack*, and its *purpose*. The intuition is that, for repeated call stack traces, these attributes should be identical, while nonrepeated call stack traces should rarely, if ever, have identical attributes. In our experiment, we have manually checked 480 call stack traces and we did not find the nonrepeated call stack traces have all these same identical attributes including IP, data type, and length. Nevertheless, even if multiple distinct call stacks have all these same identical attributes, it is also easy to optimize the efficient comparison in our work; we could add more features such as "the key packages used in the call stack" to build a more robust feature vector of call stack.

As a result, our dynamic analysis system only needs to infer the purpose of a new privacy leakage trace once. In steady state, the purposes can be reused from the cache directly, reducing the overhead of our system.

4.3. Implementation

We have implemented a prototype of our dynamic analysis approach on top of Android. Specifically, our implementation is based on TaintDroid [Enck et al. 2010] (Android Version 4.3_r1). We modified both the Android framework and Android runtime as follows:

—To construct the call stack, we modified *Dalvik* to maintain a `bridge-thread` mapping during runtime. More specifically, we instrumented and added several APIs in classes including `java.lang.Thread`, `java.util.concurrent.ThreadPoolExecutor`, and `android.os.Handler`. For example, we added four key APIs in `java.lang.Thread`, including API `setConcurrentTracingEnabled()`, API `setCallerBridge()`, API `setCalleeBridge()`, and API `getConcurrentStackTrace()`. These APIs are used to take a snapshot of the caller stack when the caller thread invokes a method to start a new thread, find the bridge-thread mapping, and concatenate the stacks together to form a full call stack.

—To infer the purpose at runtime, we implemented the library-based method and text-based machine-learning method in the *libcore* of Android. We used the SVM [2016] algorithm to do classification, and the implementation is based on LIBSVM [LibSVM 2016]. We used 460 labeled instances that use location permission provided by our static approach (Section 3) to train a classifier offline and ported it to Android.

—We use TaintDroid for taint tracking. We instrumented each taint sink point to infer the purpose based on the call stack.

4.4. Evaluation

4.4.1. Dataset. We performed experiments on 830 popular apps, including 400 popular apps randomly selected from the top 10,000 Google Play apps[6] and 430 popular apps selected from the recommendation pages of the Baidu App Market (a popular third-party market in China). We used the Monkey testing tool [Monkey 2016] to dynamically test these apps in an automated way on a Nexus 4 phone with an instrumented Android 4.3_r1 OS. Each app was tested for 60 seconds, although this can be increased easily. We performed our experiments outdoors with network accesses, in order to have the device connect to the GPS and trigger the sensitive behavior of mobile apps. Note that dynamic analysis relies heavily on the coverage of execution traces, thus it is almost impossible to reach 100% with automated testing techniques. In this work, we only focus on using dynamic analysis to infer the purpose of permission use, thus using other UI automated testing tools is outside the scope of this article.

We first evaluate the *accuracy* of our dynamic analysis system in terms of purpose inference. Next, we evaluated the *performance overhead* as compared to native Android 4.3 as well as TaintDroid.

4.4.2. Dataset Statistics. We found a total of 81 apps (out of a total of 831 apps we tested) that leak GPS location data to remote servers, 630 apps leak the IMEI, and only three apps leak the contacts. In our evaluation, we focused on the leakage of location data, because few apps (only three apps) leak contacts data in our dataset.

During our experiments, we collected 480 call stack traces that leak location, of which 171 were unique. In other words, more than 60%[7] of the call stack traces were repeated (i.e., apps tried to send sensitive data multiple times during experiments). Among the 171 unique call stack, 74 of them (more than 40%) were constructed using *thread-pairing method*, which means that they contain call stack traces from at least two threads, thus demonstrating the utility of our thread-pairing method.

4.4.3. Accuracy of Inferring Purpose. To measure the accuracy of our system, we manually checked the 171 unique call stack traces and labeled their purposes. Note that, for the permissions used by third-party libraries (e.g., ads, analytics), we could get very accurate data in our evaluation and it is easy for us to verify the detection results, because we use LibRadar [2016], an obfuscation-resilient tool developed by our team, which could accurately detect third-party libraries used in these apps based on the results of analyzing 1.2 million Android apps, even if they are obfuscated. For the call stack traces related to permission use in custom code, we used the app description, screenshots, and the text of the call stack, related decompiled code to label these purposes. We also intercepted the outgoing data at taint sinks in the Android system to try to understand the contents and the outgoing IP address they sent. Then we compared the result with the purposes our system inferred at runtime. Note that we could not label the purposes of 18 instances in our dataset, because the code is either fully obfuscated or the app mostly used native methods by calling "java.lang.reflect.Method.invokeNative." This left us with 153 unique call stack trace instances.

Overall Result. The overall result is shown in Table XII. Without considering the fully obfuscated instances, for the 153 instances, we can correctly infer the purpose of 138 instances. Considering the repeated call stacks in our dataset, we could achieve an accuracy of 94.73% (line XV, row VII in Table XII). Taking the fully obfuscated ones also into account, our overall accuracy of inferring the purpose correctly is around 80% and 90% for the unique stack traces and overall traces, respectively.

[6] Note that some apps use Google services that are inaccessible in China, thus these apps cannot run properly.

[7] Note that the longer the testing time, the higher the repetition rate.

Table XII. The Result of Inferring the Purpose of Location Permission Use at Runtime

Purpose	#Unique Call Stacks	#Correct Inferred (Unique)	%Correct Inferred (Unique)	#All Call Stacks	#Correct Inferred (All)	%Correct Inferred (All)
ad library	93	89	95.70%	234	229	97.86%
map library	3	3	100%	107	107	100%
social networking	2	2	100%	3	3	100%
analytics library	1	1	100%	8	8	100%
game engine library	1	1	100%	2	2	100%
total (library)	100	96	96%	354	349	98.59%
nearby searching	9	8	88.89%	31	29	93.55%
map and navigation	3	3	100%	15	15	100%
tracking	3	3	100%	6	6	100%
transportation	11	7	63.6%	12	8	66.67%
customization	27	21	77.78%	37	24	64.86%
total (custom code)	53	42	79.25%	101	82	81.19%
obfuscated/cannot infer	18	-	-	25	-	-
total (w/o obfuscated)	153	138	90.20%	455	431	94.73%
total (with obfuscated)	171	138	80.70%	480	431	89.80%

Results for Third-Party Libraries. Over 60% of call stacks in our evaluation are due to third-party libraries, most of which are ad libraries. Our system could achieve over 96% accuracy in inferring purposes for unique call stack traces and more than 98% for all traces. However, because the list of labeled third-party libraries [Lin et al. 2012] is incomplete, our system missed four instances in our experiment. For example, the ad library “`net.miidi`” was not labeled in the list. However, it is easy to add more labeled libraries to improve accuracy.

Results for Custom Code. For the 53 call stack traces related to permission use in custom code, we were able to infer the purpose correctly for 42 of them (79.25%). For the “map/navigation” and “tracking” purposes, we achieve 100% accuracy. For the “transportation” purpose, we only achieve an accuracy of 63.6%. The accuracy is determined by the machine-learning classifier we used. As we discussed in Section 3, two factors play an important role in the classification: distinctive features and the number of features.

4.4.4. Performance Evaluation. Since our system is implemented based on TaintDroid, our performance evaluation consists of two parts: (1) the overall system overhead using Java benchmarks, and (2) the additional performance overhead of our dynamic analysis system compared to TaintDroid.

Java Microbenchmark. We use the CaffeineMark 3.0 benchmark [CaffeineMark 2016] for Android to evaluate the performance of our system. Figure 7 compares the performance of our dynamic analysis system with TaintDroid and native Android 4.3, in terms of the CaffeineMark benchmark score.

The result shows that our system performs similar to TaintDroid (within the measurement uncertainties), since these benchmarks do not leak sensitive data. The *loop* benchmark experiences the greatest overhead, with a slowdown of about 47%. For other benchmarks, the overhead ranges from 15% to 38%. The *overall* result is the cumulative score across other individual benchmarks. *Our system has a 27% overhead with respect to unmodified Android*, primarily due to the taint tracking overhead introduced by TaintDroid.

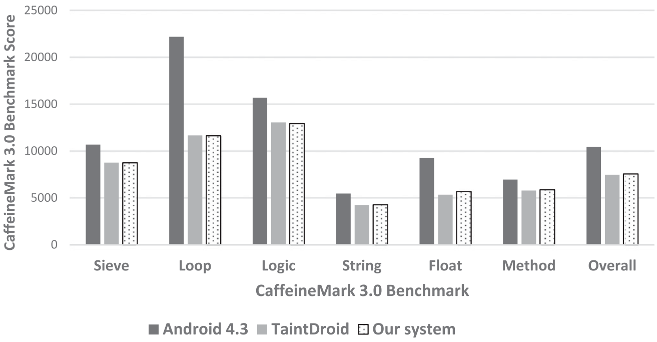


Fig. 7. Overhead of Java benchmarks when comparing our dynamic analysis system with native Android and TaintDroid (higher score is better).

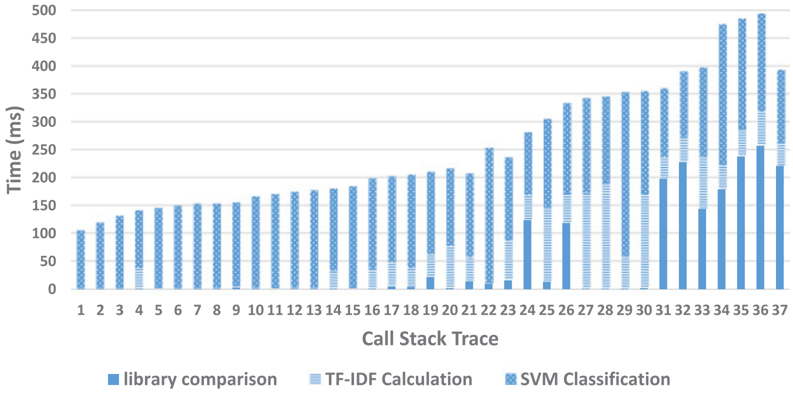


Fig. 8. Performance overhead distribution. The average performance overhead is about 258ms in total. SVM classification and TF-IDF calculation account for most of the overhead.

4.4.5. Overhead of Purpose Inferring at Runtime. Compared to TaintDroid, our system introduces overhead only when an app leaks sensitive data. The overhead imposed by our dynamic analysis system comprises four components: *call stack construction*, *library comparison*, *TF-IDF calculation*, and *SVM classification*. For apps that have no sensitive permissions, the performance of our system is the same as TaintDroid.

To measure the overhead, we instrumented the OS to log the execution time of purpose inference at the time when app leaks location data. We conducted an experiment with 30 apps and collected 253 logs (call stack traces), including 77 unique call stack traces. For the 77 unique call stack traces, 40 call stack traces used location in ad libraries, and 37 call stack traces used location in custom code. For the 40 call stack traces with the purpose of advertisement, the overhead is 53ms on average, which only contains the execution time of call stack construction and library comparison. For the 37 call stack traces that used sensitive data in custom code, the distribution of performance overhead is shown in Figure 8. The average performance overhead is about 258ms in total. For each step, the average performance overhead and standard deviation is shown in Table XIII. SVM classification accounts for most of the overhead, with an average time of 160ms. TF-IDF calculation takes 43ms on average, with a standard deviation of 29.7, which is based on the number of features (key words). The time of library comparison varies from 1ms to around 250ms, which goes up along with the increasing of call stack size. We leave out the call stack construction time in Figure 8,

Table XIII. Performance Overhead Breakdown

	Call Stack	Library	TF-IDF	SVM
Average time (ms)	5.81	49.03	43.38	159.95
Standard deviation	0	80.53	29.7	18.38

because it only costs 5.81ms on average, which is too short to compare with the other steps.

Efficacy of Purpose Caching. As mentioned earlier, some apps send the same data repeatedly, resulting in the same call stack traces. To evaluate the efficacy of our caching optimization, we analyzed the overhead of the 176 repeated call stack traces. The average time to look up the "purpose cache" is only 4.5ms, which greatly reduces the overhead of our system in steady-state operation. The result shows that *our system introduced minimal performance reduction compared with TaintDroid*.

5. COMPARISON OF THE STATIC AND DYNAMIC APPROACHES

Here we compare the static approach and the dynamic approach, discussing the pros and cons of both approaches and the trade-offs involved. First, we present a *quantitative analysis* on the static and dynamic approaches. We applied them to the same dataset and compared their performance. Then, we present a *qualitative analysis* of the static and dynamic approaches in Table XV from these aspects: *granularity* to infer the purpose, *accuracy*, *scalability*, code *coverage*, impact of code *obfuscation*, and the best fit *application scenarios*.

5.1. Quantitative Analysis

In this comparison, we manually collected more than 100 apps that likely access location data. We used several keywords to search on Google Play (e.g., "location," "nearby," "navigation," "weather," etc.), and downloaded top related apps.

We used the Monkey testing tool [Monkey 2016] to dynamically test these apps in an automated way on a Nexus 4 smartphone with an instrumented Android 4.3 r1 OS. Each app was tested for 60 seconds. We performed our experiments outdoors with network accesses, in order to have the device connect to the GPS and trigger the sensitive behavior of mobile apps. *We found 24 apps leaked GPS location data at runtime.* Note that some apps cannot run properly on Nexus 4 due to incompatible versions, and some apps use services that are inaccessible in China. To make this a fair comparison, we applied static analysis on the 24 apps to infer the purpose of permission use. Besides, to measure the effect of multithreading call stack construction, we also use the dynamic approach without multithreading call stack construction to test these apps and compare the results. We manually checked the dynamic call stack traces, and we also checked the packages that use location permission identified by static analysis to measure the accuracy of both approaches.

The result is shown in Table XIV. Note that each instance (call stack or code package) will receive 10 similarity values indicating the probabilities it belongs to each of the 10 categories (besides third-party libraries), and the sum of all 10 similarity values is equal to 1. We choose the category with the largest similarity value as its category if the similarity is larger than 0.20, otherwise we will put this instance into a new category called ***cannot infer***.

Based on the results, we make the following observations:

—*Our dynamic approach could identify the purpose of permission use in third-party libraries correctly.* For 14 apps, our dynamic approach identified the sensitive data leaked by third-party libraries, while our static analysis cannot identify these cases. Although we could extend our static approach to work on third-party libraries,

Table XIV. Quantitative Analysis of Our Static Approach and Dynamic Approach

	Dynamic Analysis			Static Analysis	
App Name	Purpose (Dynamic)	Purpose (Dynamic w/o multi)	Manually Checked	Purpose (Static)	Manually Checked
com.apalon.weatherlive.free	customized, ads(mopub)	customized, ads(mopub)	customized, ads(mopub)	customized	customized
com.aws.andoid	customized, ***geosocial***	customized, ***geosocial***	customized	customized	customized
com.local.places.near.by.me	nearby searching	***cannot infer***	nearby searching	***cannot infer***	nearby searching
com.grabtaxi.passenger	map library (mapquest), transport	map library (mapquest), transport	map library (mapquest), transport	transport	transport
air.byss.mobi.instaplacefree	analytics (flurry), ***cannot infer***	analytics (flurry), ***cannot infer***	analytics (flurry), ***cannot infer***	***cannot infer***	geotag
com.appon.mancala	ads(mopub)	ads(mopub)	ads(mopub)	***none***	***none***
com.fitnesskeeper.-runkeeper.pro	ads (KiipSDK)	ads (KiipSDK)	ads (KiipSDK)	**transport**	***cannot infer***
com.grupoheron.worldclock	ads(mopub)	ads(mopub)	ads(mopub)	customized	customized
com.reliancegames.-singhamreturnsthegame	ads(vserv)	ads(vserv)	ads(vserv)	location-based game	location-based game
com.devexpert.weather	ads(domob), customized	ads(domob), customized	ads(domob), customized	customized	customized
com.android.game3dpool	game engine (unity3d), ***social networking***, ads (crazymedia)	game engine (unity3d), ***cannot infer***, ads (crazymedia)	game engine (unity3d), ***cannot infer***, ads (crazymedia)	***cannot infer***	***cannot infer***
com.digcy.mycast	customized	***cannot infer***	customized	customized	customized
com.myteksi.passenger	***nearby searching***	***nearby searching***	transport	***nearby searching***	transport
com.raycom.kcbd	ads	ads	ads	***none***	***none***
com.tranzmate	***geosocial***	***geosocial***	transport	***geosocial***, transport	transport
com.opensignal.weathersignal	customized	***cannot infer***	customized	***cannot infer***	***cannot infer***
com.gau.go.launcherex	ads	ads	ads	***cannot infer***	***cannot infer***
com.gpsserver.gpstracker	tracking	tracking	tracking	tracking	tracking
com.gamecastor.nearbyme	social (foursquare)	social (foursquare)	social (foursquare)	***none***	***none***
air.byss.instaweather	customized	customized	customized	customized	customized
ro.startaxi.android.client	transport	***cannot infer***	transport	transport	transport
com.seatosoftware.mapapic	analytics (flurry)	analytics (flurry)	analytics (flurry)	***none***	***none***
sinhhuynh.map.fakelocation	map library	map library	map library	***none***	***none***
com.foreca.android.weather	customized	customized	customized	customized	customized

third-party libraries always contain unused permissions [Stevens et al. 2012; Wang et al. 2015b] and some third-party libraries request sensitive data by invoking methods in the app logic that provides access to resources, rather than accessing resources directly [Liu et al. 2015]. Thus, extending the static approach to work on third-party libraries could introduce false positives.

—*Our dynamic approach reconstructing call stacks across multiple threads is better than our approach without this reconstruction.* For example, the app “com.local.places.near.by.me” used the “com.android.volley” library to send asynchronous HTTP requests, thus dynamic approach without multithreading call stack construction cannot get useful information at the taint sinks, so it cannot infer the purpose as a result. Our dynamic approach could construct the full call stack traces, which could infer the purpose of the indirect data access. Besides third-party libraries, our dynamic approach could infer the purpose of permission use in custom code that static approach cannot identify in two cases.

—*Our static approach focused on the use of sensitive data (taint source), while our dynamic approach focused on the leakage of sensitive data (taint sink).* In this experiment, static analysis identified sensitive permission uses in four cases, but dynamic analysis did not find these leakages at taint sinks. For example, app “com.grupoheron.worldclock” and app “com.reliancegames.singhamreturnsthegame” were found using location permission and static approach could accurately infer the purpose, but dynamic approach did not find these leakages of sensitive data. This result indicates that static approach and dynamic approach are suitable for different usage scenarios; we will discuss it further in Section 5.2. *Besides, dynamic analysis relies heavily on the coverage of execution traces. Although static analysis has good coverage, some sensitive API calls may never be executed by the app.*

5.2. Qualitative Analysis

5.2.1. Granularity. The goal of our static approach is to identify packages that use sensitive permissions and label the purpose for each package (directory). This is based on the assumption that a directory will also have only a single purpose for a given permission. Specifying purpose at a package granularity is coarse-grained as there may be multiple purposes of data use in each package in reality. While in our dynamic approach, the purpose is determined by the call stack traces of each sensitive date leakage, which is more fine-grained and accurate.

5.2.2. Accuracy. Our static approach achieved high accuracy in our labeled dataset. However, our labeled dataset is not comprehensive. For a few apps (less than 10%) in the experiment, we could not understand how permissions are used, thus we did not use them in our evaluation of static approach. In the static approach evaluation, our dataset also did not include some apps that have unusual design patterns for using sensitive data. For example, some apps provide services that access sensitive data, while other parts of the app access these services to use sensitive data. Take the social networking app “Skout” as an example. It 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. There was very little meaningful text information in the directory where these services are located, so the static approach would simply fail.

Our dynamic approach uses fine-grained call stack traces, which could deal with this design pattern easily. By analyzing the call stack traces, we can learn which classes and methods access the sensitive data and how that data is used. *Thus our dynamic approach is more accurate than the static approach.* For the cases that our dynamic approach fails, the static approach would fail too.

Table XV. A Comparison of Our Static Approach and Dynamic Approach for Inferring Purposes in Smartphone Apps

	Static Approach	Dynamic Approach
Granularity	Coarse-grained Package level (a directory of source code)	Fine-grained (call stack trace of a sensitive data leakage)
Accuracy	Medium (cannot handle indirect permission use)	High
Scalability	High	Low
Coverage	High	Low
Application Scenarios	Market level app analysis, help respect to privacy	Purpose-based access control

5.2.3. Scalability. Our static approach does not need to run the app, which means it has good potential for scalability. In contrast, our dynamic approach is not as scalable, as it relies on dynamic testing tools to trigger an app's behaviors. Due to the limitation of automated UI testing tools, it is hard to apply dynamic analysis to millions of apps.

5.2.4. Code Coverage. While our static approach has good code coverage, our dynamic analysis approach relies heavily on execution traces, making it hard to reach complete coverage due to the large number of potential paths. Prior studies have proposed techniques for more advanced testing of mobile apps, such as UI fuzzing [Hu and Neamtiu 2011] and targeted event sequence generation [Jensen et al. 2013], which can be leveraged in our dynamic analysis in the future. It also demonstrated that the dynamic approach is suitable for privacy enforcement at runtime, rather than dynamic testing that relies on the coverage of execution traces.

5.2.5. Application Scenarios. Since our static analysis based approach has good code coverage and scalability, it is feasible to deploy it on the app market to identify sensitive behaviors of mobile apps, and help users to understand permissions used by an app and help to respect privacy. Prior work [Lin et al. 2012] showed that purpose information is important to assess people's privacy concerns. Both users' expectation and the purpose of why sensitive resources are used have a major impact on users' subjective feelings and their trust decisions. Besides, properly informing users of the purpose of resource access can ease users' privacy concerns to some extent. Shih et al. [2015] showed similar findings. They found that the purpose of data access is the main factor affecting users' privacy choices. Thus, it is important to understand the purpose of permission use and our work is the first attempt to infer the purpose of permission use from decompiled code.

Our dynamic approach is fine-grained and accurate, thus it is more suitable to deploy dynamic approach on real users' phones and help them enforce privacy protection. For example, users could define their privacy policies first, which specify whether an app is allowed to use a sensitive data item for a particular purpose (e.g., disallow accurate location for advertisement). If the detected sensitive behavior violates the policy, an exception would be thrown to block the data path. Based on our experiment, the overhead of inferring purpose at runtime is negligible and imperceptible to mobile users. The average performance overhead to infer the purpose of sensitive data use is 258ms at runtime. Using a purpose caching optimization, the overhead is reduced to 4.5ms on average in steady state.

5.3. Purpose-Based Access Control

To demonstrate the usability of our dynamic analysis, we have implemented a prototype access control system that can enforce purpose-based privacy policies. As shown in

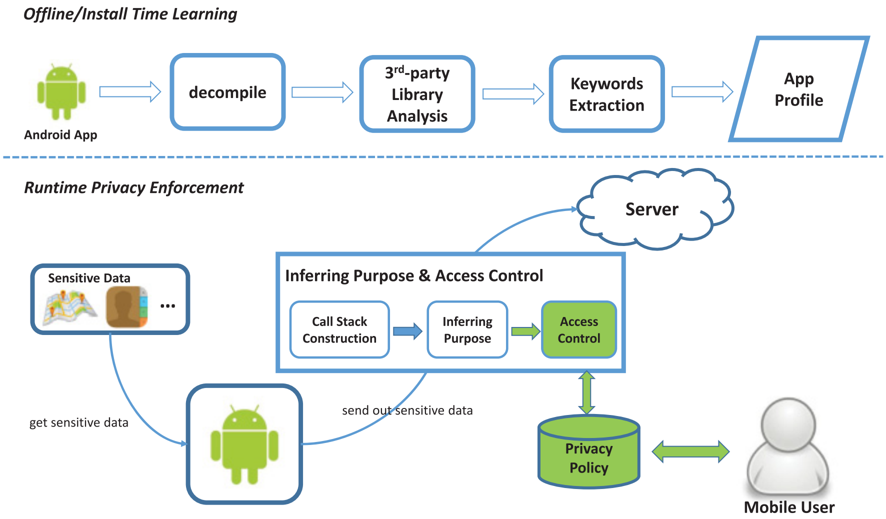


Fig. 9. Overall architecture of the prototype access control system. We added the *privacy policy* and *access control* parts (padding with green) based on the dynamic analysis framework we proposed.

Table XVI. Examples of Access Control Policies

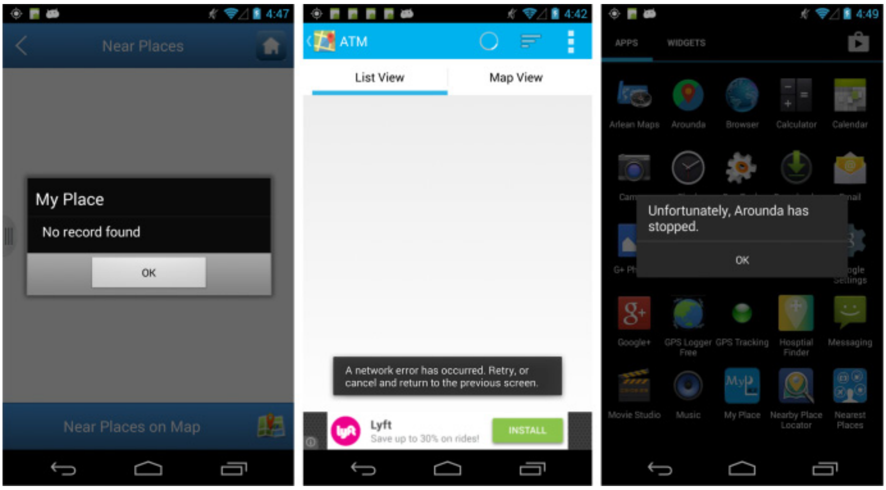
Policy	Description
$< location, ads, block >$	disallow accurate location for advertisement
$< location, nearbysearching, allow >$	allow to use location for nearby searching

Figure 9, we added the *privacy policy* and *access control* parts based on the dynamic analysis framework we proposed.

Users can easily define global privacy policies for all the apps using a triple <`permission, purpose, action`>. For example, a set of privacy policies for a particular user could use the form as shown in Table XVI. Further, we expect that more complex policies can also be implemented on top of our system in the future. For example, user could define policies based on *app category*, *app name*, *used permission*, *purpose of permission use*, *destination IP address*, and *whether it uses SSL connection*. For example, a user could block egress of sensitive contacts data for all game apps. Furthermore, we could use context information such as *at home* or *at work* to enforce purpose-based context-aware access control.

Note that currently we do not have a UI to specify these policies for our prototype system. Instead, in this article we focus on exploring the capability of dynamic analysis in inferring purposes, and enabling the new functionality of purpose-based control, and demonstrating its feasibility. We leave the design and evaluation of appropriate UIs for allowing users to specify these access policies to future work. However, we note that such a UI can be integrated with Android AppOps or with other systems such as the ProtectMyPrivacy app [Agarwal and Hall 2013].

For policy enforcement, we modified TaintDroid such that at each sink point the app behavior is checked against user-defined policies. If the sensitive behavior violates the policy, an exception would be thrown to block the data path. Note that if the app does not catch and handle the exception, the app may crash. Our goal is to let users selectively enforce privacy policies for sensitive behaviors associated with certain purposes, without affecting other behaviors or functionalities of the app. During our experiments, we observed three kinds of results for blocking sensitive data at runtime, as shown in Figure 10.


My Place

No record found

A network error has occurred. Retry, or cancel and return to the previous screen.

Unfortunately, Arounda has stopped.

Fig. 10. Impact to app functionality. Examples of three kinds of behaviors if we block sensitive data: "run normally and no result is shown," "run normally but show error," and "app crash at runtime."

Blocking necessary sensitive data use in some apps can cause it to crash (less than 10% of apps in our experiment), mainly because the apps did not catch and handle the exceptions when our system blocked the data. In contrast, blocking sensitive data in third-party libraries rarely caused crashes. We also note that since the arrival of fine-grained permission control in Android 6.0, it is only a matter of time before developers will change their apps to add exceptional handlers as users use the Android UI to allow or deny access to sensitive data to the entire app.

6. DISCUSSION

6.1. Code Obfuscation

In our previous experiments, we first identified the classes that use sensitive permissions, then we determined whether the class is obfuscated or not. We only examined code using permission-related android APIs, and we found that about 10% of apps contain obfuscated code, with much of it belonging to third-party libraries. Previous research [Linares-Vásquez et al. 2014] analyzed 24,379 Android apps, and they only found 415 apps (less than 2%) with obfuscated custom code.

To further measure the code obfuscation rate in current Android apps and measure the effectiveness of our approach, we manually downloaded 1,600 popular Android apps from Google Play in September 2016. All of them are top apps from different categories. Then we analyzed these apps in detail.

We focus on four research questions:

—How many of the popular apps are obfuscated?
—How many of them are fully obfuscated? Even when an app is obfuscated, not all classes in it are obfuscated (e.g., some code cannot be obfuscated because it is defined or referenced externally, etc). So what is the obfuscation rate of these obfuscated apps?
—Do they have significant impact on the effectiveness of our approach?
—Are there any feasible ways to deal with code obfuscation?

We investigated these apps in detail, and answer each question in the following.

6.1.1. How Many of the Popular Apps are Obfuscated? Following previous work [Linares-Vásquez et al. 2014], we use a simple heuristic to measure whether an app is obfuscated or not. This heuristic is based on the fact that certain obfuscators, in particular the popular tool Proguard, renames classes using a lexicographic order. Therefore, to detect obfuscated apps, we look for apps with class names that have only a single letter, for example, a.java, b.java, c.java, etc. We decided to use this simple heuristic because we were interested only in the impact of identifier obfuscation. *That is to say, as long as we find an app with a class named with a single letter, we will mark this app as obfuscated.*

For the 1,600 popular apps, 1,144 of them are marked as obfuscated apps, which accounts for 71.5% of the apps. This result suggested that obfuscation is quite popular in Android apps. But does it mean we cannot infer the purpose of permission use in these apps? We further analyzed these apps in the following.

6.1.2. How Many of Them are Fully Obfuscated? What is the Obfuscation Rate of Obfuscated Apps? Note that even if an app is obfuscated, not all classes in it are obfuscated. On one hand, some code cannot be obfuscated because it is defined or referenced externally, such as APIs defined in the framework and components related to the Android app lifecycle. On the other hand, some code may need extra efforts if they are to be obfuscated. For example, some complicated packages or classes may result in runtime errors due to improper ProGuard rules. Many developers would leave these packages and classes alone because they have to debug them and configure detailed obfuscation rules if they want to obfuscate them.

We define *obfuscation rate* as the proportion of likely obfuscated classes (a class in which more than 50% of the identifier names are likely obfuscated) among all classes in an app. We build an identifier name dictionary to identify regular obfuscated names, including the names in short alphabet format (e.g., a, b, c, aa, ab,...) produced by ProGuard in default setting and other customized rules using different dictionaries.

As a result, we find that most of the obfuscated packages and classes are from third-party libraries, while the obfuscation rate in custom code is low. Roughly more than 50% of the obfuscated apps have obfuscation rate less than 20% in their custom code excluding third-party libraries. Only 14 apps (out of 1,600 apps we examined) are fully obfuscated.

6.1.3. Do they have Significant Impact on the Effectiveness of Our Approach? We use an obfuscation-resilient method [LibRadar 2016; Ma et al. 2016] to identify third-party libraries in the app based on Android API features. Most of the obfuscated classes are from third-party libraries, so these classes almost have no impact on the effectiveness of our approach.

For code obfuscation in custom code, as long as they are not fully obfuscated, our approach might still be able to extract meaningful features and learn its purpose. Excluding third-party libraries, most of the apps do not have a higher obfuscation rate.

We also examined the apps we studied in our previous experiment. For the roughly 600 apps in our static analysis, around 300 of them are found to have a class that is named with a single letter, which means roughly 50% of them are possibly obfuscated. But in our previous experiment, we could still label the purposes and using text-mining to extract features and learn the purposes.

Thus, whether code obfuscation could have great impact on the effectiveness of our approach depends on the obfuscation level and obfuscation rate.

6.1.4. Are there any Feasible Ways to Deal with Code Obfuscation? A recent work DEGUARD [Bichsel et al. 2016] was proposed to reverse layout obfuscation (naming obfuscation) of Android APKs. In layout obfuscation, the names of program identifiers that carry key semantic information are replaced with other (short) identifiers with no semantic meaning. Examples of such elements are variable, method, and class names. They learn probabilistic models from “Big Code” and then use these models to achieve overall precision and scalability of the probabilistic predictions. It could recover 79.1% of the program element names obfuscated with ProGuard, which could be used in our work to recover obfuscated code and help us extract meaningful features.

In summary, based on our preliminary study on 1,600 recent popular apps from Google Play, we have the following findings:

—Code obfuscation is quite popular in Android apps; more than 70% of apps are obfuscated to some extent in our study.
—Most of the obfuscated packages and classes are from third-party libraries, while the obfuscation rate in custom code is low. Only 14 apps (out of 1,600 apps we examined) are fully obfuscated.
—Third-party library obfuscation almost has no impact on the effectiveness of our approach. Whether code obfuscation could have great impact on the effectiveness of our approach depends on the obfuscation level and obfuscation rate of custom code.
—There are some feasible ways to deal with code obfuscation, which could be potentially used to help us infer the purpose.

6.2. Implicit Control Flow and Native Code

Our dynamic analysis system inherits two limitations from TaintDroid, that is, *implicit control flow analysis* and *native code issues*. TaintDroid does not track implicit dataflows, for example, an app’s control flow [Sarwar et al. 2013] (e.g., conditional branching). Besides, native code is unmonitored in TaintDroid. Thus, our dynamic analysis approach would fail in these cases. Subsequent work [Gilbert et al. 2011] proposed to add implicit flow support to TaintDroid, which we could use to improve our system.

6.3. Indirect Permission Use

As stated earlier, some apps use sensitive data through a level of indirection rather than directly accessing it. In this case, our static analysis approach would fail, while our dynamic approach could deal with this design pattern easily. 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 multilabel classification.

6.4. ICC-Based Multihreading

The thread-pairing method we used to construct the full call stack at runtime is also able to handle the case of Inter-Component Communication (ICC) based multithreading. Using ICC, the parent thread can send an intent to framework, and the framework handles the intent to start a new thread. In this case, the “intent” object can be used as a bridge between the sender thread and receiver thread, just like the “task” object used as a bridge between caller thread and callee thread in AsyncTask-based multithreading. Figure 11 shows an example of ICC-based multithreading, the sender Activity starts the receiver Activity by sending an Intent, and the “intent” object is shared by both sender and receiver. We can hook the “startActivity()” method in sender thread to record the mapping from sender to the intent, and hook the “onCreate()” method of receiver to get the “intent” object that started the receiver thread.

Thus, it is easy to extend our current dynamic analysis system and implement ICC-based call stack construction. Previous work AppContext [Yang et al. 2015] proposed to chain all ICCs within the app and construct an Extended Call Graph (ECG) to infer activation events, which we could also use to improve our work. We did not

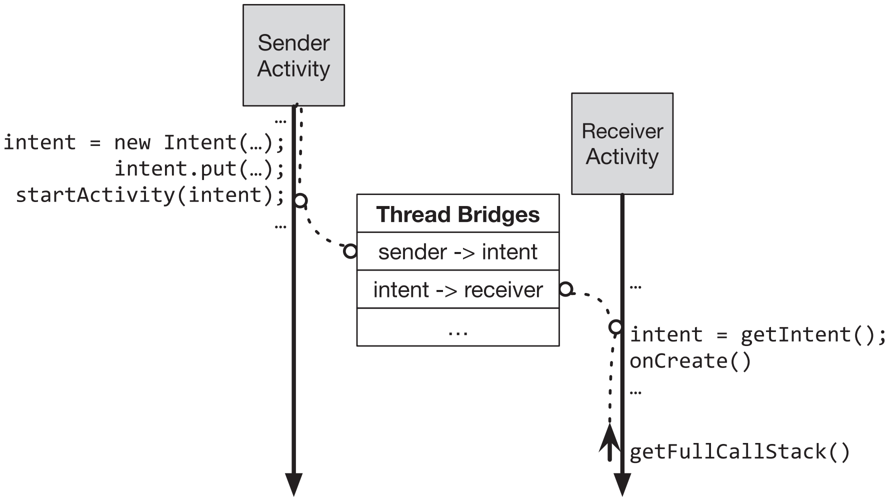


Fig. 11. A bridge-building example of ICC-based multithreading. The "intent" object can be used as a bridge between the sender thread and receiver thread.

implement it in the current system, because ICC is often used to start an Android component (Activity, Service, etc.). In this article, we think different components often have different purposes. For example, a normal Activity may start a new Activity to present an Advertisement. As our goal is to infer the purpose based on call stack traces, we only need the call stack of the current Android component. Although it is better to connect current component with the background services in some cases (e.g., malware performs suspicious behaviors in the thread initiated by ICC), we are still not sure how much it will impact the performance of our system. During our preliminary experiment, we found that if current component is connected with other components using ICC and they cooperate to exhibit some behaviors, the current component will need to receive intent from the other components, and the code that handles the intent will provide some information to help us infer the purpose of permission use in the current component. We will further analyze this issue in the future.

6.5. 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 [StringMatching 2016] to infer abbreviated words.

6.6. Expanding to Other Permissions and Purposes

We have created a taxonomy of 10 purposes for the location permission and 10 purposes for the contacts permission. While our taxonomy is good enough for our experiments, 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. This article demonstrated that we could infer purpose from the decompiled code or call stack at runtime. 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.

Besides, if possible, and depending on how the purposes are used, we could use clustering-based approaches to automatically learn the purposes of permission uses from the extracted text features in future work. For example, one possible way is that we could use LDA on the extracted texts from decompiled permission-related code, and identify the main topics for each package, and then cluster packages by related topics. We could regard each cluster as a “purpose” of permission use. Based on how the purposes are used, we could use clustering algorithm such as k-means to define the number of clusters. Then we could identify fine-grained or coarse-grained “purposes” based on the number of clusters. Note that one problem remains here is that maybe it is hard to assign a name for each automated identified purpose.

Moreover, previous work AppContext [Yang et al. 2015] proposed to use information flow analysis and machine learning to identify malicious behaviors, which we could use to improve our work and identify malicious purposes.

6.7. Bypassing Our Detection System

Note that our work 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.

6.8. Practicality and Usability of the Dynamic System

The goal of this article is to show that purpose-based access control of permissions is indeed possible and to present a prototype implementation. In order to deploy our dynamic system widely to regular users, we will ideally need the functionality we have proposed to be integrated into the OS itself (e.g., Android or through a port such as Cyanogen) and support different versions of Android. Our work is based on TaintDroid to track sensitive information flow, which only supports up to Android 4.2. To work on new versions of Android (especially 6.0 and above), we should use other dynamic taint analysis approaches.

Furthermore, while prior work showed that purpose information is important to assess people’s privacy concerns, there have been no user studies to show how users interact with a system with these capabilities and what the appropriate UI might look like. We are investigating ways to deploy and test our system on real users, but note that it will require an extensive user study.

7. CONCLUSIONS

In this article, we propose a text mining based method to infer the purpose of a permission use for Android apps. We present the design, implementation, and evaluation of two approaches to inferring purposes, which are based on static analysis and dynamic analysis, respectively. We first evaluate the effectiveness of using text analysis techniques on decompiled code statically. Our experiments show that we can achieve about 85% accuracy in inferring the purpose of location use, and 94% for contact list use. Then we introduce a dynamic analysis technique to overcome the limitations of static analysis. For the dynamic approach, we try to infer the purpose of permission use in the entire app, including third-party libraries and custom code. Experimental results show that we are able to successfully infer the purpose of over 90% sensitive location data uses. We also discuss the pros and cons of both static and dynamic approaches, and the trade-offs involved.

REFERENCES

Yuvraj Agarwal and Malcolm Hall. 2013. ProtectMyPrivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceedings of the 11th Annual International Conference on Mobile Systems, Applications, and Services*. 97–110.

Hazim Almuhimedi, Florian Schaub, Norman Sadeh, Idris Adjerid, Alessandro Acquisti, Joshua Gluck, Lorrie Faith Cranor, and Yuvraj Agarwal. 2015. 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)*. 787–796.

Shahriyar Amini, Jialiu Lin, Jason I. Hong, Janne Lindqvist, and Joy Zhang. 2013. Mobile application evaluation using automation and crowdsourcing. In *Proceedings of the PETools*.

Apktool 2016. Apktool: A tool for reverse engineering Android apk files. Retrieved from https://code.google.com/p/android-apktool/.

AppStore 2016. Wikipedia *App Store (iOS)*. Retrieved from https://en.wikipedia.org/wiki/App_Store_(iOS).

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. 259–269.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 217–228.

Michael Backes, Sven Bugiel, Sebastian Gerling, and Philipp von Styp-Rekowsky. 2014. Android security framework: Extensible multi-layered access control on android. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*. 46–55.

Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard: Enforcing user requirements on android apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. 543–548.

Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. 2013. "Little brothers watching you": Raising awareness of data leaks on smartphones. In *Proceedings of the 9th Symposium on Usable Privacy and Security (SOUPS'13)*. 12:1–12:11.

Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*. 49–54.

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical deobfuscation of android applications. In *CCS'16*.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. 131–146.

C4.5 2016. Wikipedia. *C4.5 Algorithm*. (2016). http://en.wikipedia.org/wiki/C4.5_algorithm.

CaffeineMark 2016. CaffeineMark. Retrieved from https://play.google.com/store/apps/details?id=com.android.cm3&hl=zh_CN.

Erika Chin, Adrienne Porter Felt, Vyas Sekar, and David Wagner. 2012. Measuring user confidence in smartphone security and privacy. In *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS'12)*.

Cross-Validation 2016. Wikipedia. *Cross-validation*. Retrieved from https://en.wikipedia.org/wiki/Cross-validation_(statistics).

Benjamin Davis and Hao Chen. 2013. RetroSkeleton: Retrofitting android apps. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*.

Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. 2012. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. In *Proceedings of the Mobile Security Technologies*.

Dex2jar 2016. dex2jar. Retrieved from https://code.google.com/p/dex2jar/.

Serge Egelman, Adrienne Porter Felt, and David Wagner. 2012. Choice architecture and smartphone privacy: There's a price for that. In *Proceedings of the Workshop on the Economics of Information Security (WEIS)*.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. 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)*.

William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*.

William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. 235–245.

Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS'12)*. 3:1–3:14.

Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. 2011. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the 2nd International Workshop on Mobile Cloud Computing and Services (MCS'11)*. 21–26.

GooglePlay 2016. Wikipedia. *Google Play*. Retrieved from http://en.wikipedia.org/wiki/Google_Play.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information flow analysis of android applications in DroidSafe. In *Proceedings of NDSS 2015*.

Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. 2014. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 1025–1035.

Marian Harbach, Markus Hettig, Susanne Weber, and Matthew Smith. 2014. 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)*.

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A programmable interface for extending android security. In *Proceedigns of the 23rd USENIX Security Symposium (USENIX Security'14)*. 1005–1019.

Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. 639–652.

Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*. 77–83.

Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. 106–117.

Qatrunnada Ismail, Tousif Ahmed, Apu Kapadia, and Michael Reiter. 2015. Crowdsourced exploration of security configurations. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'15)*.

JD-Core-Java 2016. JD-Core-Java. Retrieved from http://jd.benow.ca/.

Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of ISSTA'13*. 67–77.

Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. 2014. 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)*. 99–110.

Jaeyeon Jung, Seungyeop Han, and David Wetherall. 2012. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*. 45–50.

Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. 2013. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'13)*. 3393–3402.

LibRadar 2016. LibRadar: Detecting Libraries in Android Apps. Retrieved from http://radar.pkuos.org/. (2016).

LibSVM 2016. LIBSVM—A Library for Support Vector Machines. Retrieved from https://www.csie.ntu.edu.tw/ cjlin/libsvm/.

Jialiu Lin, Shahriyar Amini, Jason I. Hong, Norman Sadeh, Janne Lindqvist, and Joy Zhang. 2012. Expectation and purpose: Understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing (UbiComp'12)*. 501–510.

Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I. Hong. 2014. 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)*.

Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. 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)*. 242–251.

Bin Liu, Bin Liu, Hongxia Jin, and Ramesh View. 2015. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the the 13th International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*.

Looper 2016. Looper. Retrieved from http://developer.android.com/reference/android/os/Looper.html.

Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion*. 653–656.

Mallet 2016. *Mallet:* MAchine Learning for LanguagE ToolkiT. Retrieved from http://mallet.cs.umass.edu/.

Clara Mancini, Keerthi Thomas, Yvonne Rogers, Blaine A. Price, Lukazs Jedrzejczyk, Arosha K. Bandara, Adam N. Joinson, and Bashar Nuseibeh. 2009. From spaces to places: Emerging contexts in mobile privacy. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp'09)*. 1–10.

Maximum Entropy 2016. Wikipedia *Maximum Entropy*. Retrieved from http://en.wikipedia.org/wiki/Maximum_entropy.

Monkey 2016. UI/Application Exerciser Monkey. Retrieved from developer.android.com/tools/help/monkey.html.

MultipleThreads 2016. MultipleThreads. Retrieved from http://developer.android.com/intl/en-us/training/multiple-threads/index.html.

Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10)*. 328–332.

Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2009. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC'09)*. 340–349.

Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. 2013. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. 527–542.

Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. AdDroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS'12)*.

PermissionMappings 2015. Permission mappings. Retrieved from http://pscout.csl.toronto.edu/.

Porter 2015. The Porter Stemming Algorithm. Retrieved from http://tartarus.org/martin/PorterStemmer/.

PrivacyGrade 2015. PrivacyGrade: Grading the privacy of smartphone apps. Retrieved from http://privacygrade.org/.

PScout API 2015. Documented API calls mappings. Retrieved from http://pscout.csl.toronto.edu/download.php?file=results/jellybean_publishedapimapping.

PScout ContentProvider 2015. Content Provider (URI strings) with permissions. Retrieved from http://pscout.csl.toronto.edu/download.php?file=results/jellybean_contentproviderpermission.

PScout Intent 2015. Intents with Permissions. Retrieved from http://pscout.csl.toronto.edu/download.php?file=results/jellybean_intentpermissions.

Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. 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)*. 1354–1365.

Franziska Roesner and Tadayoshi Kohno. 2013. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. 97–112.

Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. 2013. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *Proceedings of the 10th International Conference on Security and Cryptography (SECRYPT'13)*. 461–467.

Daniel Schreckling, Johannes Kstler, and Matthias Schaff. 2013. Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for Android. *Information Security Technical Report* 17, 3 (2013), 71–80.

SciKit 2016. *Scikit-learn* Machine learning in Python. Retrieved from http://scikit-learn.org/stable/index.html.

Shashi Shekhar, Michael Dietz, and Dan S. Wallach. 2012. AdSplit: Separating smartphone advertising from applications. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*.

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

Irina Shklovski, Scott D. Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. 2014. 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)*. 2347–2356.

Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *Proceedings of the Workshop on Mobile Security Technologies (MoST)*.

StringMatching 2016. Wikipedia *Approximate String Matching*. Retrieved from http://en.wikipedia.org/wiki/Approximate_string_matching.

SVM 2016. Wikipedia *Support Vector Machine*. Retrieved from http://en.wikipedia.org/wiki/Support_vector_machine.

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. 77–91.

Eran Toch, Justin Cranshaw, Paul Hankes Drielsma, Janice Y. Tsai, Patrick Gage Kelley, James Springfield, Lorrie Cranor, Jason Hong, and Norman Sadeh. 2010. Empirical models of privacy in location sharing. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp'10)*. 129–138.

Omer Tripp and Julia Rubin. 2014. A Bayesian approach to privacy enforcement in smartphones. In *Proceedings of the 23rd USENIX Conference on Security Symposium (Security'14)*.

Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015a. WuKong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA'15)*. 71–82.

Haoyu Wang, Yao Guo, Zihao Tang, Guangdong Bai, and Xiangqun Chen. 2015b. Reevaluating android permission gaps with static and dynamic analysis. In *Proceedings of GLOBECOM'15*.

Haoyu Wang, Jason I. Hong, and Yao Guo. 2015c. Using text mining to infer the purpose of permission use in mobile apps. In *The 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'15)*. 1107–1118.

Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. 2017. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proceedings of the 26th International Conference on World Wide Web (WWW'17)*. 163–172.

Jiayu Wang and Qigeng Chen. 2014. ASPG: Generating android semantic permissions. In *Proceedings of the IEEE 17th International Conference on Computational Science and Engineering*. 591–598.

Takuya Watanabe, Mitsuaki Akiyama, Tetsuya Sakai, and Tatsuya Mori. 2015. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *11th Symposium On Usable Privacy and Security (SOUPS 2015)*. 241–255.

WordList 2015. English wordlist. (2015). http://www-personal.umich.edu/jlawler/wordlist.

Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*.

Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. AppContext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. 303–313.

Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*. 1043–1054.

Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. 2011. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*. 93–107.

Received July 2016; revised March 2017; accepted April 2017