Android Apps Consistency Scrutinized

Khalid Alharbi
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
khalid.alharbi@colorado.edu

Atif Memon
Department of Computer Science
University of Maryland
College Park, MD, USA
atif@cs.umd.edu

Sam Blackshear
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
samuel.blackshear@colorado.edu

Bor-Yuh Evan Chang
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
evan.chang@colorado.edu

Emily Kowalczyk
Department of Computer Science
University of Maryland
College Park, MD, USA
ekowalcz@umd.edu

Tom Yeh
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
tom.yeh@colorado.edu

CHI 2014, Apr 26 - May 01 2014, Toronto, ON, Canada
ACM 978-1-4503-2474-8/14/04.
http://dx.doi.org/10.1145/2559206.2581352

Abstract
The increasingly larger selection of mobile apps has made it difficult for users to understand what a particular app does and how it differs from the others. A user typically learns about an app from the app's public information (while deciding whether to install it), from the app's UI (while exploring the UI), and from the app's actual behaviors (while using it). Users may become confused or surprised if there are inconsistencies between (a) the public information and UI, (b) the UI and the actual behavior, or (c) the public information and the actual behavior. For example, turning on the camera (actual behavior) when there is no button that says SNAP (UI) is a potentially confusing inconsistency. We present work-in-progress toward a methodology for automatically detecting inconsistencies in Android apps with respect to permissions and similarity. We report our preliminary results on a large corpus of 178,765 apps.

Author Keywords
UI; data; analysis; mobile; Android; permissions

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

Introduction
There are over a million Android apps available on various app markets. The Google Play Store, the official

and largest market, has more than a million apps. Android uses a permission system to protect device's resources and user's data. There are many apps that ask for unexpected permissions. Why does this app ask for the camera permission but never mentions a word about it? Why does this app ask for the camera permission but never invokes any API method to operate the camera? Is this a bug, malice, or a feature? These are potential inconsistencies that often confuse users.

Within a group of apps, there may exist some apps significantly different from the rest. As an example, consider a set of apps grouped together because they all require camera permissions. After analyzing a large number of these apps, we may spot a common pattern exhibited by the majority of these apps. The pattern may be that an app's public description tends to contain words such as capture, take picture, or photo. A user might be confused when he finds an app that says nothing publicly about using cameras even though it explicitly requires camera permissions. Inconsistency may occur between various levels. Consider an app whose description includes words like take a picture but it doesn't include anything at the UI level. Additionally, the UI may include "take a picture" button, yet that button doesn't trigger the phone's camera. As a result of this inconsistency, users may be surprised, annoyed, confused, or even harmed. We studied two questions regarding inconsistencies in Android apps:

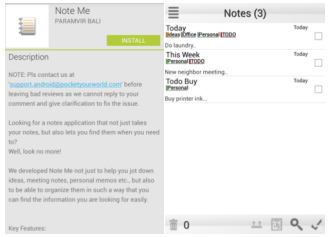
1. How can we compute a descriptor to provide a comprehensive and comparable view of an app?
2. How can we apply such a descriptor to discover inconsistencies between an app's public features, UI features, and code features?

To address these questions, we took a data-driven approach. We collected a corpus of 178,765 Android apps from the Google Play Store. We computed descriptors of Android apps by combining features extracted at three levels. At the public level, we considered an app's online description, package name, category, ratings, install size, and other publicly available information. At the UI level, we considered an app's user interface's text and layout. At the code level, we considered the methods declared, invoked and the connections between event handlers (e.g., onClick) and library calls (e.g., takePicture), using program analysis. We present preliminary findings on real cases of inconsistences discovered by our methodology with respect to camera permissions and similarity. This methodology can produce results beneficial to several stakeholder groups: (a) End users can use the results to find apps that are consistent with their permissions. (b) Developers can use these results to resolve the inconsistency in their apps. (c) Marketplace owners can use our methodology and add it to their existing vetting process for submitted apps.

Related Work

Previous research efforts on Android permission system have studied the effectiveness of the permissions display interface and users expectations for permissions [3,4]. While it's important to inform users about the application's security permissions, generally, users pay less attention to the permissions display and have low rates of understanding their meanings [2].

WHYPER [6] uses NLP semantic analysis of an app's description and finds sentences that describe need for permissions. Our methodology is comprehensive and considers a set of public information, UI, and the



Note Me

Public Features:

Description:	Looking for a notes application that not just takes your notes... (208 words)
Permissions:	android.permission.VIBRATE, android.permission.INTERNET ... (6 permissions)
Others	Title: Note Me, Creator: Paramvir Bali, InstallSIze: 559085, Rating: 4.325572, Category: Productivity, ContentRating: Everyone

User Interface Features:

Text	Settings QuickNotesMain Note Me Search Enter Title Note Text Title Created On (3411 words)
Layout	ScrollView$export_options_view LinearLayout$note_edit (26 files)

Code Features:

App	createNote getSomeTitleForNote deleteNote (1875 methods)
API	activityOnDestroy launchAdActivity getApplicationContext (4034 methods)
Points-to	Lcom/quicknotes/views/**NoteView**$7, **onClick** → Landroid/database/sqlite/SQLiteDatabase,**query** (4912 pairs)

Table 1: Example of features extracted at three levels to form a descriptor for an app (Note Me).

source code at large scale. We focus on analyzing the possible interactions a user might have with an app with respect to permissions. This includes the user actions that are supposed to occur before, during, and after installing a new app.

Android App Descriptor

We begin our analysis of inconsistencies in Android apps by computing a descriptor for each app. The role of a descriptor is to give a comprehensive and comparative view of an app. A descriptor should comprehensively capture as many aspects about an app as possible. It should also allow efficient and effective comparison. We propose a novel descriptor composed of features extracted at three levels. Table 1 shows an excerpt of the three-level features extracted to form a descriptor of an app (Note Me). Next are details on these features:

Public Features: These features are derived from public information about an app and visible to users before the app is installed. We extracted public features about an app from the details published on the Google Play Store. We used Google-Play-Crawler[1], an unofficial open source API for the Google Play Store, to download APK files, collect package names, reviews, permissions, title, creator, and number of downloads for each application from the Google Play Store. NOKOGIRI[2] was used to gather further info from each app's Google Play Store web page including: app description, category, rating, date published, Play Store URL, price, version, operating system, ratings count, content rating, developer URL, install size, and downloads count text.

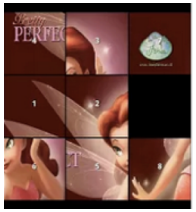
User Interface (UI) Features: These features are derived from the user interface of an app, including the text and layout. They are visible only to users who have installed and are using the app. We extracted two types of user interface features: text and layout. We downloaded each app and saved it as an APK file (Android Package). We used apktool[3], an open source reverse-engineering tool, to unpack the APK file into a directory tree of program files that make up the app. To extract layout features, we looked into the res/layout directory for layout files. We parsed them all using a custom XML parser and collected all the widgets we encountered as features. To extract text features, we parsed three string resources files, (strings.xml, arrays.xml, and plurals.xml). We extracted strings from layout files when strings are hardcoded into layout files.

Code Features: These features are derived from the disassembled and decompiled code of an app. We are interested in the following code features: (a) an app's own methods, (b) the Android library methods an app invokes, and (c) the connection between a user event handler (e.g., onlick) and the method triggered by the event (e.g., takePicture). These features are not visible to users but can be inspected by an expert program analyst to examine the app's actual behavior. We extracted three types of code features: declared methods, invoked methods, and pairs. We used apktool to unpack an app's APK file. We located the file classes.dex, which bundles the binary code of all the classes of an app into a single file. We used smali to

[1] https://github.com/Akdeniz/google-play-crawler

[2] http://nokogiri.org

[3] https://code.google.com/p/android-apktool



TinkerBell Puzzle

Public: Fun & Addicting Puzzle Game Create Puzzles from TinkerBell Photos Challenge & Compete with your friends ...
User Interface: Tinkerbell Puzzle All photos are collected from the search engines ... Take Picture ... New Game High Scores Settings New Best Time ...

Figure 1: Example of Public/Interface Inconsistency.

Animals Game for Kids

User Interface: Balloon Animals for Kids Menu button to resume NEW GAME RESUME G ... (71 words)
Code: org/anddev/andengine/engine/camera/Camera;->onUpdate org/anddev/andengine/engine/camera/Camera;-><init>

Figure 2: Example of Interface/Code Inconsistency.

disassemble classes.dex into individual files, one per class, in a human-readable assembly-like format. To extract declared method features, we looked for the pattern .method [declared name] in every smali file and collected all occurrences. To extract invoke method features, we looked for the pattern .invoke-virtual and .invoke-public. To extract method pair features, we built a call graph [7] for each app using the WALA[4] program analysis framework. WALA takes Java classes.dex into Java bytecode. Our call graphs are 1-object-sensitive with unlimited context sensitivity for container classes.

Preliminary Dataset

We collected our own corpus of Android apps. This corpus consists of 178,765 apps published on the Google Play Store. We extracted the public features of all 178,765 apps to perform our analysis. Among these apps, 153,294 were free. We took a sample of 84,405 apps (about 50%). We downloaded, unpacked, disassembled, and decompiled them. Each app yielded about 1,000 program files we must process. From these files, we extracted user interface and code features. We stored all extracted features using MongoDB, a NoSQL database optimized for big data analysis. The total size of raw data is about 4TB.

Camera Permission Inconsistencies

In our corpus of 178,765 apps, we found 17,739 (9.9%) apps requiring camera permissions. At the public level, we used all these apps as positive examples. We randomly selected the same number of apps that do not require camera permissions as negative examples. We trained a model for positive camera permission apps based on Maximum entropy [1]. The training accuracy was 98.7%. We then applied the classifier to the 17,739 apps that require camera permissions. Among these apps, 307 were classified as not requiring camera permissions. These apps present inconsistency that could not fit the model. At the user interface level, we analyzed a subset of 7,816 apps requiring camera permissions. We repeated a similar training process. The training accuracy was 92.9%. 498 were classified as inconsistent. Next discusses three types of camera inconsistencies we discovered.

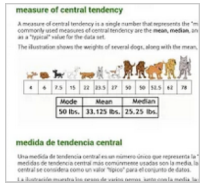
Inconsistency between Public (P) and Interface (I)

An app is inconsistent between its public information and interface if the interface presents certain sensitive features that are not disclosed on the app's public page in the app market. TinkerBell Puzzle (Figure 1) is an example we discovered that exhibits this type of inconsistency. It is a puzzle game. The app's public description mentions "photos" but it does not suggest the photo is being taken or shot by cameras. At the interface level, the combination of the word "photos" and the phrase "Take a Picture" provides strong evidence to the camera use. In this case, users may be confused since the app's description does not clearly describe the camera feature. Consequently, they may avoid installing the app to explore the UI.

Inconsistency between Interface (U) and Code (C)

An app is inconsistent between its user interface and code if the label of an interface component (e.g., "New") does not match the code this component invokes (e.g., takePicture). Animals Game for Kids (Figure 2) is an example we discovered that exhibits this type of inconsistency. It is a game for kids. There is no indication on the user interface that the camera is

[4] http://wala.sourceforge.net



MathTerms mathterms.com.andyfelong

Public: MathTerms is an illustrated glossary of mathematics terms in English and Spanish....
Code: invoke-static {}, Lti/modules/titanium/media/TiCameraActivity;->takePicture()V

Figure 3: Example of Public/Code Inconsistency.

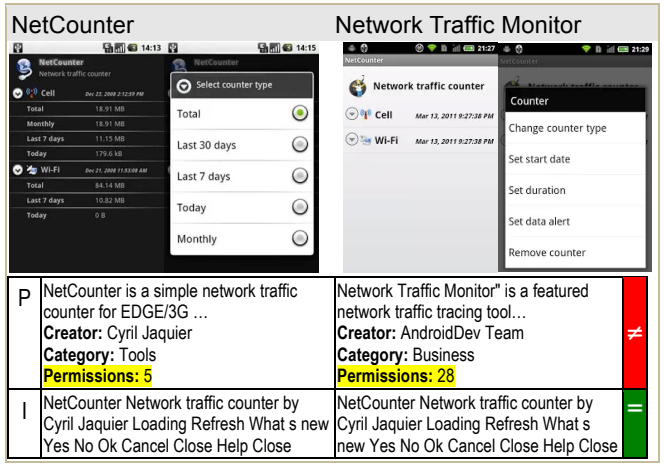
used. But at the code level, camera API calls are found. There is no logical connection between any interface component and these API calls. In this case, users would be confused because it is not clear what UI component triggers the camera function.

Inconsistency between Public (P) and Code (C)

An app is inconsistent between the public and the code levels when there is a mismatch between its public description and its actual behavior as revealed by code. Figure 3 shows an actual example of this type of inconsistency we discovered. This app appears to be a glossary app. The app's code contains calls to take pictures. A user would be very surprised when a picture is taken while he/she is using the app to look up math terms.

Similarity Inconsistencies

A classic example of similarity inconsistency is when two apps appear to be different at the public level but have very similar user interfaces. We further analyze our dataset to identify this type of inconsistency. We used the classic term frequency-inverse document frequency (*tf-idf*) model to calculate the similarity among apps. We used 95% as the threshold. If two apps are more than 95% similar, they are considered as near identical. We treat them as inconsistent apps because most apps have no near identical apps. At the public level, we analyzed 11,880 apps and focused on just the app's description. We found 630 apps (5.3%) with near identical apps in terms of their description. Let's denote the set of these apps as P. At the interface level, we analyzed a sample of 72,993 apps and found 1,339 apps (1.83%) with near identical apps. Let's denote this set as I. Having found P and I, we can compute the *difference* between the two, which will tell us which apps are similar at one level but not at the other.



P	NetCounter is a simple network traffic counter for EDGE/3G ... **Creator:** Cyril Jaquier **Category:** Tools **Permissions:** 5	Network Traffic Monitor" is a featured network traffic tracing tool... **Creator:** AndroidDev Team **Category:** Business **Permissions:** 28	≠
I	NetCounter Network traffic counter by Cyril Jaquier Loading Refresh What s new Yes No Ok Cancel Close Help Close	NetCounter Network traffic counter by Cyril Jaquier Loading Refresh What s new Yes No Ok Cancel Close Help Close	=

As an example, NetCounter and Network Traffic Monitor are a pair of apps we discovered that exhibit similarity *not* at the public level and *only* at the user interface level. By reading the public information, users would see them described differently, made by separate creators, and filed under distinct categories. But our UI similarity analysis reveals that the interfaces of the two apps are almost identical, suggesting that one app may be a knockoff of the other. An unsuspecting user would have no way to tell until after installing the app. A more cautious user may compare the two apps and notice that one requires as many as 28 permissions while the other requires only five. Also, one has more than 20K user ratings with an average of 4.5 star while the other has only 177 ratings with an average of 2.5 stars. Based on this comparison, one may deduce that the one that receives weaker ratings and requires more permissions is probably a knockoff. Unfortunately, Google Play's similarity calculation does not take into

account the user interface or the code. These two apps are not listed as similar apps to allow users to compare.

Conclusion and Future Work

In this paper, we presented a work-in-progress toward a new approach to identifying inconsistencies in Android apps. Our preliminary results show the promise of applying comprehensive and automated analysis techniques on Android apps. Our preliminary findings of inconsistency are only related to camera permissions and similarities. We are currently extending this methodology to other types of inconsistences, such as

- Other types of permissions (e.g., location, network)
- Statistics (e.g., Number of Requested Permissions, Install Size, User Ratings)
- Content ratings (e.g., general, mature)
- Advertisement (e.g., free, ad-supported)
- Category labels (e.g., tools, references, games)

In addition, we are aware of a number of limitations we must address as future work, including:

- Approach: Our code features extraction has some inherent limitations to static analysis tools. Although we only used dex2jar to decompile apps to Java bytecode for the pair of methods analysis, previous work has shown that dex2jar fails in certain cases [5]. We need to assess to what degree this affects the accuracy of our approach.
- Evaluation: We need to systematically evaluate our approach on two key metrics: accuracy and success rate at the code feature extraction level.
- Image elements are not analyzed. An app may use graphics to provide visual cues to users (e.g., camera icon to start the camera), which are not included in our analysis. We need to explore solutions using computer vision.

Acknowledgement

This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0039. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Berger, A.L., Pietra, V.J.D., and Pietra, S.A.D. A Maximum Entropy Approach to Natural Language Processing. *Comput. Linguist. 22*, 1 (1996), 39–71.
2. Felt, A.P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android Permissions: User Attention, Comprehension, and Behavior. In *SOUPS*, 2012.
3. Kelley, P.G., Cranor, L.F., and Sadeh, N. Privacy As Part of the App Decision-making Process. In *CHI*, 2013.
4. Lin, J., Amini, S., Hong, J.I., Sadeh, N., Lindqvist, J., and Zhang, J. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy Through Crowdsourcing. *In Ubicomp*, 2012.
5. Octeau, D., Jha, S., and McDaniel, P. Retargeting Android Applications to Java Bytecode. In *SIGSOFT*, 2012.
6. Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: towards automating risk assessment of mobile applications. In *USENIX Security*, 2013.
7. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., and Yahav, E. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, 196–232.