Checking App User Interfaces against App Descriptions

Konstantin Kuznetsov
Saarland University
Saarbrücken, Germany
kuznetsov@cs.uni-saarland.de

Vitalii Avdiienko
Saarland University
Saarbrücken, Germany
avdiienko@cs.uni-saarland.de

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain
alessandra.gorla@imdea.org

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Does the advertised behavior of apps correlate with what a user sees on a screen? In this paper, we introduce a technique to statically extract the text from the user interface definitions of an Android app. We use this technique to compare the natural language topics of an app's user interface against the topics from its app store description. A mismatch indicates that some feature is exposed by the user interface, but is not present in the description, or vice versa. The popular Twitter app, for instance, spots UI elements that allow to make purchases; however, this feature is not mentioned in its description. Likewise, we identified a number of apps whose user interface asks users to access or supply sensitive data; but this "feature" is not mentioned in the description. In the long run, analyzing user interface topics and comparing them against external descriptions opens the way for checking general mismatches between requirements and implementation.

CCS Concepts

•Human-centered computing → **HCI design and evaluation methods; •Software and its engineering** → *Software libraries and repositories;* **•Information systems** → Document topic models; **•Computing methodologies** → Anomaly detection;

Keywords

Android; App mining; Topic models; UI Anomalies;

1. INTRODUCTION

When a user decides whether to install an Android application or not, she does not know much about its actual behavior. A brief description and a set of screenshots give a high level intuition of features that the app has. However, a description that does not fully represent an advertised functionality can confuse a user.

The description of a well-known TWITTER[1] application offers a full range of services: stay informed with breaking news, share them with friends and even communicate via direct messages. Indeed, when a user launches the app and logs-in a list of TWITTER posts ("tweets") appears on the screen. The user interface (UI) prompts to discover new events, find friends, or send a private message exactly as advertised. However, a curious user while exploring the app may stumble across a "Orders and payments" menu item, which leads to a surprising "Order history" screen (Figure 1). Is it really possible to buy goods or functionality with the TWITTER app? Unfortunately, its description does not mention this functionality at all.

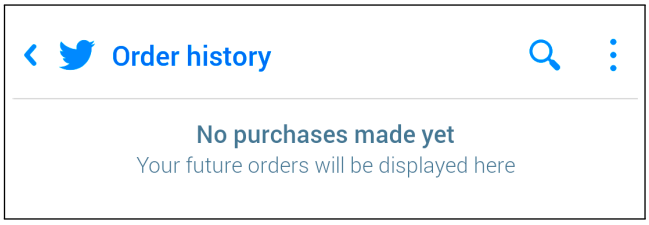

Figure 1: Order history screen from TWITTER

We found that for many apps, developers fail to provide detailed information on what an application is doing [3, 5]; over time, a description also may no longer comprise the latest evolution of the app's functionality. Even worse, a user might open an app's page and find only a few words. As an example, the popular "Snapchat"[2] app has just one line in its description "*Life's more fun when you live in the moment :) Happy Snapping!*" which is not that helpful. Only the later note "*Snapchatters can always capture or save your messages, such as by taking a screenshot or using a camera. Be mindful of what you Snap!*" gives a little idea of what the app is actually doing.

In this work, we attempt to check *whether the advertised behavior of apps from the market correlates with what users see on the screen* and, if not, suggest changes to improve the matching of a description against the user interface. To the best of our knowledge, this is the *first fully automated approach to detect mismatches between descriptions and user interfaces;* a technique only made possible by the ability to automatically mine both the user interfaces of apps as well as their market meta data.

Our work starts by mining the TOP100 applications of each category from the Google Play Store, as well as all available apps from the F-Droid open source repository (more than 1800+ apps). For each app, we collect both the user interface data (as part of its APK

[1]https://play.google.com/store/apps/details?id=com.twitter.android

[2]https://play.google.com/store/apps/details?id=com.snapchat.-android

package), as well as the description data (as part of its app market metadata). We then apply topic modeling to the corpus of descriptions, and later use the inferred topic model on the user interface data. Thus, for each app we obtain two sets of probabilities of it belonging to each inferred topic: the first set of probabilities refers to its description, and the second one to its user interface. We finally compare, for each app, the description topic distribution with the UI topic distribution, and we report mismatches.

In the remainder of this paper, we present each step of our technique in detail. In Section 2 we discuss how to collect text from app's UI; to the best of our knowledge, this is the first work systematically exploiting and summarizing this data source on a large scale. After describing how we access app descriptions (Section 3), we continue with Section 4 and describe the construction of the topic model. In Section 5 we explain how we identify mismatches in topics, the main novel contribution of this work. In Section 6 we report examples of mismatches between descriptions and UI artifacts. After an analysis of related work (Section 7). we close with conclusion and future work (Section 8).

2. MINING ANDROID USER INTERFACES

The *user interface* of an app is everything that the user can see and interact with. For this reason, user interface elements such as buttons and menus usually rely on self-explanatory text labels to describe their functionalities.

We thus analyze, as a first step of our technique, the text that the user interface contains, and we use it as a proxy to describe the underlying functionalities that UI elements would trigger.

The user interface in Android apps is a hierarchy of `View` and `ViewGroup` objects organized in a composite pattern:

Views. A `View` is a base object for all UI elements in Android. A `View` is an object that draws something on the screen that the user can interact with.

ViewGroups. A `ViewGroup` is a *View* that is responsible for holding other `View` or `ViewGroup` objects and defines the structure of the layout.

Developers can define the layout of the screen in two ways:

1. They can declare the UI elements and their layout within an XML file. As an example, consider Figure 2, which shows how the UI elements of the TWITTER app shown in Figure 1 are arranged in a vertical linear layout. These XML files come as part of the ANDROID app resources, and are part of the app's APK package file.

2. Alternatively, they can programmatically declare the hierarchy of UI elements in the Java code of the app. This is rarely done, however, because the XML alternative is easier and provides a separation of concerns.

Regardless of how they define layouts, developers usually provide some *text* to describe UI elements. They can bind text to UI elements either by using the `android:text` attribute in the XML layout file or programmatically at runtime. The text, in turn, can be either a reference to the app's resources or can be the string that will be displayed directly. However, the last approach is not very used in practice, because it makes localization hard to deal with. Text resources are thus typically specified in `strings.xml` files—one default and other ones for each desired language.

As an example, consider Figure 3, showing the `strings.xml` file related to the XML layout file in Figure 2. The XML layout file includes a `TypefacesTextView` widget whose `android:text`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout>
 <com.twitter.ui.widget.TypefacesTextView
  android:id=
   "@id/commerce_payment_shipping_loading"
  android:text=
   "@string/commerce_history_no_history"/>
 <com.twitter.ui.widget.TypefacesTextView
  android:text=
   "@string/commerce_history_no_history_subtitle"/>
</LinearLayout>
```

Figure 2: TWITTERXML layout file (excerpt) defining the UI of Figure 1.

```
<?xml version="1.0" encoding="utf-8"?>
 <resources>
  <string name="commerce_history_no_history">
    No purchases made yet
  </string>
  <string name="commerce_history_no_history_subtitle">
    Your future orders will be displayed here
  </string>
 </resources>
```

Figure 3: Localized text in the TWITTER app (excerpt) defining strings used in Figure 2.

feature is `"@string/commerce_history_no_history"`. The file `strings.xml` then defines this string as `"No purchases made yet"`, the default string to be shown (as seen on Figure 1), unless the localization provides a language-specific replacement.

In this work we assume that most of the apps follow guidelines and use resource files to store text.

As a consequence of our assumption, we analyze all –but only– UI resources files of apps to extract text that is associated with their user interfaces. As an example, the analysis of the screen in Figure 1 produces the following bag of words:

```
Order history, no purchases made yet, your future
     orders will be
displayed here
```

After removing duplicates, we process UI text –we follow the same process for descriptions, as we describe in the following Section– with the standard natural language processing techniques of filtering, lemmatization and stemming as follows:

1. Given a bag of words, we first remove all non-text items such as numerals, URL links and e-mail addresses.

2. Then, using the Language Detection Library for Java [8], we detect the most likely language of each phrase and remove all non-English words.

3. Next, we remove stop words, i.e., common words such as "a", "to", "by", which do not carry meaningful information. We enriched the common stop word list for English language with our own domain specific set of words. We created this new list of terms by applying the baseline approach described in [6] on the description and UI corpora separately, and taking the top 100 stop words from the results.

4. To further reduce the words in the corpora, we also remove rare words, which appear less then twice.

5. Finally, we apply *lemmatization* to reduce the inflectional forms of a word to a common base form using a vocabulary and morphological analysis. It is essential to make words such as "send", "sent", and "sending" all match to the single dictionary base form "send".

Figure 4: Words from UI elements of TWITTER

Figure 5: Words from TWITTER description

6. We use stemming to additionally strip inflectional and derivational suffixes from terms. So, "sender" turns into "send". In essence, lemmatization and stemming help reducing the number of words in the corpus, and consequently improve the results of the following topic mining process.

Across all TWITTER screens, we thus obtain the set of words depicted in Figure 4. The most frequent words ("discard", "cancel") are generic items that would be present in most user interfaces; however "profile", "follow", and of course, "tweet", are already more specific to the TWITTER app.

3. MINING DESCRIPTIONS

Together with APK files we download the metadata details of each app in our dataset [2]. Metadata include any public information that the Google Play store displays (e.g., description, user rating and category). We then parse the metadata information to extract the app description, and we process the text with the same natural language processing techniques for descriptions as we do for user interface text—that is, filtering, lemmatization and stemming. As a result, we again obtain a bag of words, together with their frequency that represents the description of the app.

What do these descriptions look like? As an example, consider Figure 5, showing the word cloud for the description of TWITTER. A human can already spot some key differences by comparing words in the description and the words in the user interface (represented in Figure 4). However, in order to have an automated comparison and report anomalies we first *abstract* them according to the *topics* that they relate to.

4. TOPIC EXTRACTION

A careful look at the words used in the UI of TWITTER vs. the ones used in its description (reported in the central part of Figure 4 and Figure 5 respectively) can highlight some mismatches. Instead of directly comparing words, however, we compare description and UI text according to the *topics* they belong to.

We leverage LDA to build a topic model of our corpus of apps. Topic modeling provides a simple way to analyze large volumes of unlabeled text. Using contextual information, topic modeling can connect words with similar meanings as well as distinguish between uses of words with multiple meanings. The approach is build on top of the concept of "topic", which consists of a *cluster of words that frequently occur together*. Each textual document in the corpus can then be described with probabilities of belonging to each topic.

We leverage topic modeling by first training the model on the corpus of app descriptions. Table 1 shows the list of topics of our model trained on all the descriptions of *all* analyzed apps. For each topic we report the most representative words and a representative name that we manually assigned. After building the topic model on the description corpus, we use the same model to infer which topics the UI text belongs to. Applied on the TWITTER application, for instance, our technique would assign the textual description to four topics (*messaging, social, news & videos and other*), while the UI text would match five topics (*messaging, social, account, purchases and other*). In Figure 4 and Figure 5 topics are listed in the outer part, within quotes.

In order to identify topics we used the *Mallet* topic mining tool [7]. Since it is necessary to specify the number of topics to generate, we tried several values and manually evaluated the quality of the resulting topics. We found that 30 topics is a reasonable value for our corpus.

Once we have both the description and UI of each application assigned to different topics, we proceed to the next step, i.e. to find and report anomalies. The UI labels in TWITTER, for instance, reveal the *purchases* topic, which does not appear in the app description. This mismatch indicates that there are features in the UI that are not mentioned in the description. In the next Section we describe how we can automatically detect such anomalies and turn them into recommendations to update the app description such that it matches the functionalities in the app.

5. ANOMALY DETECTION

We define anomalies those applications whose UI topic distribution significantly differs from the topic distribution of its corresponding description on the market.

More precisely, for each application we compare the probability of its description and UI to belonging to each topic. Topic probabilities are values that range between 0 and 1. The sum of all probabilities for an application equals to 1. If the UI is associated with a high probability to some topic, whereas its description is not

Table 1: Topics mined from Android app descriptions

Id	Assigned Name	Most Representative Words (stemmed)
0	"photos"	**photo** effect camera filter collag **editor galleri** frame pictur sticker share
1	"soccer"	**game** play leagu score match player **footbal** challeng world team soccer win build ball level puzzl
2	"funny pictures"	**draw** anim funni face **pictur** eye share friend girl step fashion style hair charact paint
3	"settings"	**launcher** sound alarm light **theme** turn bar press switch action clock lock full flashlight
4	"subscriptions"	access user **subscript offer** premium **purchas** term updat visit store polici person content
5	"news & videos"	**news** stori **video content** watch read channel articl broadcast top stream world popular follow explor
6	"account"	card scan **contact password** send read credit address store account permiss secur encrypt custom key import
7	"purchases"	**shop** product card offer item order store price gift fashion buy reward find deliveri amazon sale brand deal search earn
8	"audio"	**music play** song player radio listen audio artist sound station album record track lyric
9	"connection"	**server** remot **access** system root sourc client user run send build medium updat librari command project
10	"wallpapers"	**wallpap** beauti background anim year amaz world christma beach waterfal flower love decor man awesom pictur water friend heart lot
11	"pregnancy"	period track **babi** fertil account **pregnanc** expens money ovul month tracker market cycl
12	"readers"	**document read** page print book browser **share** comic web format reader offic search printer bookmark
13	"dictionaries"	**languag** spanish french english portugues german chines russian italian japanes translat turkish korean
14	"personalize"	keyboard **theme** font galaxi launcher input layout smart **wallpap** key languag predict style person
15	"games"	**play** child talk charact **game** friend unlock tom life watch purchas privaci real person magic item bubbl babi anim laser
16	"diet"	**weight** food track calori diari **health** bodi **diet** healthi medic remind pill tracker lose recip water counter fit goal drink
17	"weather"	**weather forecast** clock locat temperatur hour condit max wind rain radar local humid updat world citi skin report sunset sunris
18	"calendar"	**calendar event** lock task mod remind unlock properti schedul pin birthday idea assist access project week month organ pattern import
19	"synchronization"	**search** job permiss access **backup** account store find storag sync send filter share import multipl folder restor mail move extern
20	"workout"	**workout** train fit **exercis** run person goal track sleep minut challeng heart brain plan daili bodi progress muscl perform coach
21	"navigation"	**map locat** track rout speed traffic alert find countri road warn tracker drive street unit camera world cell friend record
22	"messaging"	call **messag** friend contact **chat** group send share video famili record photo block caller instant messeng convers peopl receiv sticker
23	"sensors"	**sensor** measur pressur level system tool speed model star sound point engin sky moon move realist touch virtual finger high
24	"calculations"	**calcul** ski **value** fritz point resort line hour shift statist total report math graph averag exampl oper unit amount snow
25	"learning language"	**learn languag** word english german translat dictionari vocabulari cours speak phrase pronunci spanish listen grammar studi audio practic quiz travel
26	"media"	**video** camera record face **music** share **audio** watch photo medium movi captur stream play pictur swap clip cam moment detect
27	"social"	**find friend** peopl share love world user meet question million search follow twitter chat communiti great dont alway join good
28	"travel"	**book travel** car ticket map find bus citi hotel flight station vehicl transport search trip rout berlin price train locat
29	"system"	**batteri** secur protect clean **memori** speed power **system** safe privaci lock antivirus boost space storag block privat charg scan perform

—or it is with a much lower probability— we infer that the description lacks some important details that can be described by this topic, and we thus report this topic as *missing*. We used the value 0.2 as a threshold to report an anomaly.

The TWITTER description clearly puts this app into the *social* topic, since it mentions that it allows people to connect with each other (Figure 6). The UI text also supports this intuition, since there are buttons with labels *Sync contacts* and *Find friends* as well as other related textual hints. The probability of belonging to the social topic of its description and its UI text is thus quite high, and since the difference in the probability values is less than the threshold, we do not report TWITTER as anomalous with respect to this topic.

Along with posting tweets on a public newsboard, TWITTER allows to exchange private messages with friends. The user can open a dedicated *Messages* screen, choose a recipient and create a new message by typing it in a text field, which is labeled with a *Start a new message* tag. The description mentions this feature, and as a consequence we act as for the previous topic.

TWITTER also helps users staying informed with recent news and events. Its description is thus also associated with the *news & videos* topic. However, the text in the UI does not show this facility. Since all the tweets are dynamically generated at runtime, our analysis does not find this information in its static resources. In this

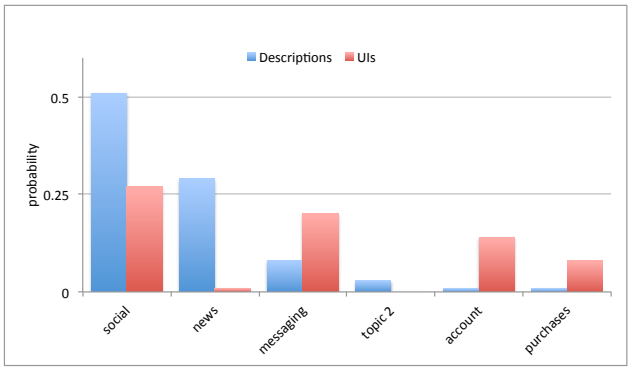

Figure 6: TWITTER topics distribution

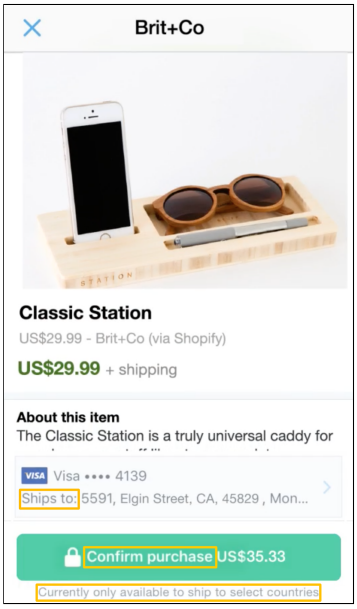

Figure 7: The TWITTER purchase service.

case, though, we do not report this mismatch, since the "missing" information is in the UI rather than in the description.

Our analysis, though, finds a significant mismatch in the probabilities related to the *purchases* topic. This topic is predominant in the UI, and it shows that there are some functionalities associated to it, while the description completely lacks any information about it. This is, thus, an anomaly that we report. The app developers could thus take the result of our approach and enrich the app description on the market to explain why and how these purchase features can be used in the app.

6. SOME MISMATCHES

We performed our analysis on the TOP100 Android applications on the Google Play Store in December 2015, and all the applications from F-Droid repository. After preprocessing and removing improper descriptions (e.g., non english ones), the dataset boiled down to 3735 apps. Here we report some of the interesting mismatches that we found thanks to our analysis.

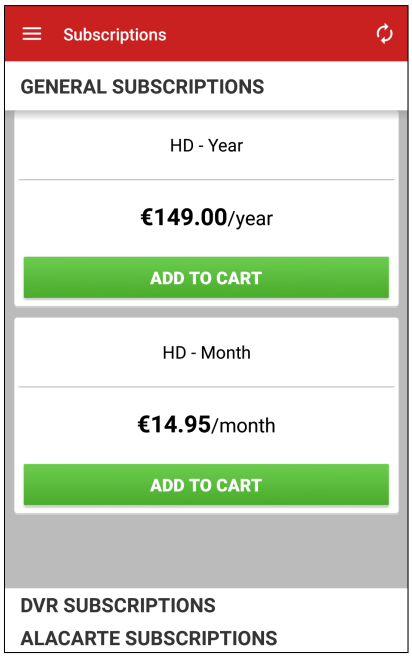

Figure 8: The *FilmOn Free Live TV* subscriptions screen.

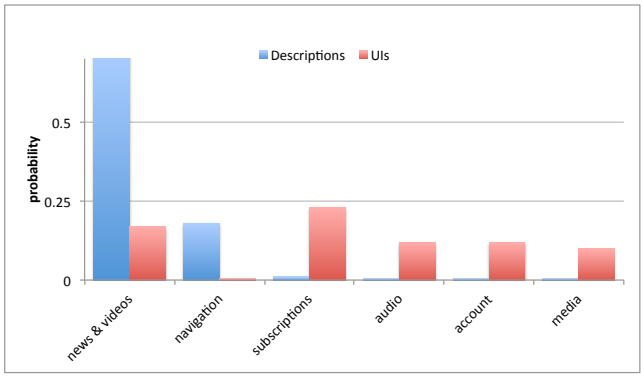

Figure 9: The *FilmOn Free Live TV* topics distribution.

In-app Purchases

TWITTER is one of the examples that we found of hidden purchases in the app. Actually, we found that its UI text has many more words that are related to the "purchases" topic, even more than what the average user can see. It turns out that these UI elements are related to the *In-Tweet* purchase service, introduced by the *TWITTER company* in September of 2014. Verified users can purchase products using TWITTER by tapping on the "Buy" button and by entering billing information (Figure 7). Nevertheless, this option is absolutely not mentioned in the app's description.

Although rumors say that TWITTER discontinued this service in May 2016 [3], by now it is still possible to find a working *"Buy"* button in rare tweets on the web-version of TWITTER. The TWITTER app that we have analyzed is of 2015, when in-app service should have been still available.

[3]https://www.buzzfeed.com/alexkantrowitz/twitter-disbands-commerce-team-ceases-product-development-on

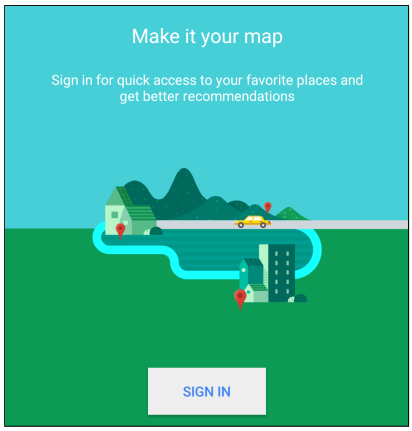

Figure 10: The *Google maps* log-in screen.

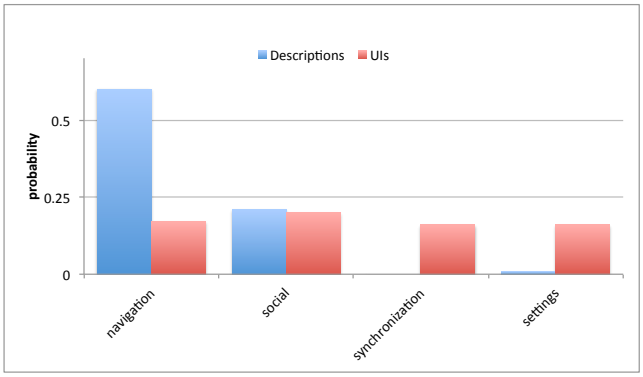

Figure 11: The *Google maps* topics distribution.

Premium Services

Many applications being free software allow users to use premium services via paid subscriptions. For instance, despite the *Free* word in its name, *FilmOn Free Live TV* provides access to paid channels, as visible in Figure 8. This app has neither any word about payments nor a specific label "Offers in-app purchases" which can be assigned by the Google Play Store. A user gets to know of this feature only by using the app. The topic distribution Figure 9 clearly identifies this discrepancy between the advertised features and actual behavior. It should be noted that this app was wrongly assigned to the "navigation" topic because the description mentions local TVs with a lot of destinations.

Account & Synchronization

In order to fully exploit an application and get access to all features, sometimes users are required to create an account. We identified many apps that ask users to set up an account and submit their private data. This behavior is especially annoying if the type of the app does not explicitly mention account creation in the description. For example, the *Google maps* app is highly integrated with the *Google* infrastructure. It can add places mentioned in e-mails, save favorite locations, and synchronize points of interest between mobile and web apps. Though, a user must be logged in with the *Google* account credentials in order to use all these features (Figure 10).

The mismatch in the UI description and the textual description is clearly visible in Figure 11. The UI has a predominant "synchro-

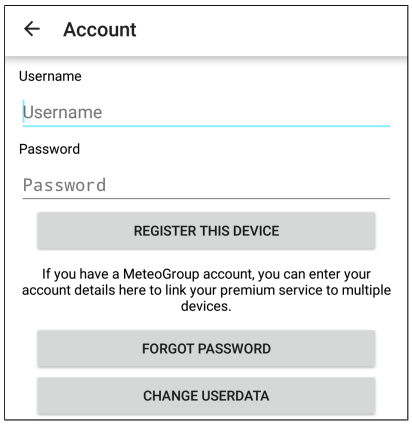

Figure 12: The *RainToday – HD Radar* log-in screen.

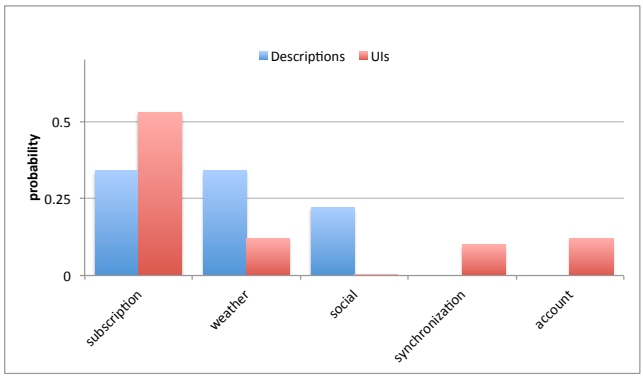

Figure 13: The *RainToday – HD Radar* topics distribution.

nization" topic, which clearly implies the necessity to log-in with a valid account. This topic is missing in the textual description, instead.

RainToday app is another example of this category of mismatches. It features real-time rain alerts and a high resolution radar. In its description they advertise a lot of modern features like modern interface and high-resolution and animated graphs. There exists a premium version, which offers ads-free functionalities.

The description of the app does not mention anything related to the premium version, and the possibility to create an account to upgrade. Even if the account is only used for linking premium services to the user, this should be at least mentioned in the description. The mismatch can be clearly see in Figure 13 in the "synchronization" topic as it was the case for Google Maps.

Location

User location is a sensitive information, and usually developers tend to explain why their apps require GPS data. In our analysis we found a wallpaper application that accesses the user's location. This immediately seemed to be dubious, if not malicious, behavior. *Christmas CM Locker Theme* is a theme for the screen locker application. With a thorough analysis, it turned out that it uses the location service to provide weather forecast information to the user. Nevertheless, this is not mentioned in its description. The developer should be careful as some users may provide a low rate to this application because of this suspicious feature.

Figure 14: The *Christmas CM Locker Theme* screen.

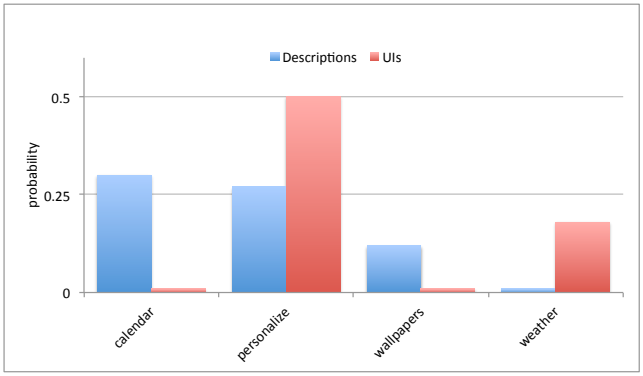

Figure 15: The *Christmas CM Locker Theme* topics distribution.

7. RELATED WORK

While this work may be the first to generally check app user interfaces against app descriptions, it builds on a history of previous work of mining app descriptions and mining app user interfaces.

Several techniques focus on detecting whether the *claimed* behavior matches the *actual* behavior of the application. Some of these techniques use the textual description to understand what an application should do. Gorla et al. [3, 5] mined apps for mismatches between textual description and used APIs while Pandita et al. [9] and Qu et al. [10] measured a correspondence between a textual description and declared permissions. Yu et al. [11] use privacy policies rather than descriptions for the same purpose, while Al-Subaihin et al. [1] applied better clustering techniques to group together similar applications in terms of their description. Huang et al. [4], in turn, analyze specific labels associated to UI elements and compare them to the invoked APIs.

The current work is orthogonal to all these techniques. It abstracts away from the nature of Android applications, and works only on the natural language processing level and measure mismatches in the *claimed* and the *actual* behavior based only on a user's experience.

8. CONCLUSION AND FUTURE WORK

The user interface of applications is a valuable and yet unexplored data source that can be used to *augment* the natural language information about an app, but also to detect *mismatches* between a

description from the app market and the actual usable functionality. At this point, we have just begun to exploit this new and exciting data set, and our initial results are more than promising. We see several opportunities where user interface data might become very useful, and our future work will focus on these topics:

Exploiting apps with insufficient descriptions. If the description of an app is short or incomplete, the UI text can be helpful to obtain additional data. This is particularly interesting for malware samples, which may not be obtained from an app market, and which therefore lack appropriate metadata.

Detecting mismatches between advertised and actual behavior. In earlier work, we had used app descriptions to detect mismatches between descriptions and API usage [3]. This work could be extended to make use of UI text—either as an addition or even an alternative to the app market text.

Programmatic UI construction. So far, we only extract user interface text from APK resource files; however, we could also apply static analysis tools to identify user interface construction code, as well as associated strings. This would be especially useful for user interface text created at runtime, such as error messages.

Linking program and user interface text. By relating program functions with the user interface elements that trigger their execution, or by relating elements with the functions that create them, we can associate program locations with natural language items and descriptions. This could open an entirely new way of reasoning about what a program function does, and whether its behavior can be considered normal.

To learn more about our work on app mining, including the data sets used in this paper, check out our Web site

https://www.st.cs.uni-saarland.de/appmining/

9. ACKNOWLEDGMENTS

This work was supported by the European Research Council, project "SPECMATE", the European Union FP7-PEOPLE-COFUND project AMAROUT II (grant n. 291803), by the Spanish Ministry of Economy project DEDETIS, by the Madrid Regional Government project *N-Greens Software* (grant n. S2013/ICE-2731), and by the EIT Digital project SMAPPER.

10. REFERENCES

[1] A. A. Al-Subaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang. Clustering mobile apps based on mined textual descriptions. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '16, 2016.

[2] V. Avdiienko, K. Kuznetsov, P. Calciati, J. C. C. Román, A. Gorla, and A. Zeller. CALAPPA: a toolchain for mining android applications. In *Proceedings of the 1st International Workshop on App Market Analytics*, WAMA 2016, pages –. ACM, 11 2016.

[3] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1025–1035, New York, NY, USA, 2014. ACM.

[4] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[5] K. Kuznetsov, A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Mining android apps for anomalies. In *The Art and Science of Analyzing Software Data*, pages 257–281. Morgan Kaufmann, 4 2015.

[6] R. T.-W. Lo, B. He, and I. Ounis. Automatically building a stopword list for an information retrieval system. In *Information Retrieval Workshop*, page 17. Citeseer, 2005.

[7] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[8] S. Nakatani. Language detection library for Java, 2010.

[9] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, pages 527–542, 2013.

[10] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, , and Z. Chen. AutoCog: Measuring the description-to-permission fidelity in Android applications. In *Proceedings of the 21st Conference on Computer and Communications Security (CCS)*, 2014.

[11] L. Yu, X. Luo, C. Qian, and S. Wang. Revisiting the description-to-behavior fidelity in android applications. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 415–426, March 2016.