Checking App Behavior Against App Descriptions

Alessandra Gorla · Ilaria Tavecchia[*] · Florian Gross · Andreas Zeller
Saarland University
Saarbrücken, Germany
{gorla, tavecchia, fgross, zeller}@cs.uni-saarland.de

ABSTRACT

How do we know a program does what it claims to do? After clustering Android apps by their description topics, we identify outliers in each cluster with respect to their API usage. A “weather” app that sends messages thus becomes an anomaly; likewise, a “messaging” app would typically not be expected to access the current location. Applied on a set of 22,500+ Android applications, our CHABADA prototype identified several anomalies; additionally, it flagged 56% of novel malware as such, without requiring any known malware patterns.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Android, malware detection, description analysis, clustering

1. INTRODUCTION

Checking whether a program does what it claims to do is a long-standing problem for developers. Unfortunately, it now has become a problem for computer users, too. Whenever we install a new app, we run the risk of the app being “malware”—that is, to act against the interests of its users.

Research and industry so far have focused on detecting malware by checking static code and dynamic behavior against predefined patterns of malicious behavior. However, this will not help against new attacks, as it is hard to define in advance whether some program behavior will be beneficial or malicious. The problem is that any specification on what makes behavior beneficial or malicious *very much depends on the current context.* In the mobile world, for instance, a behavior considered malicious in one app may well be a feature of another app:

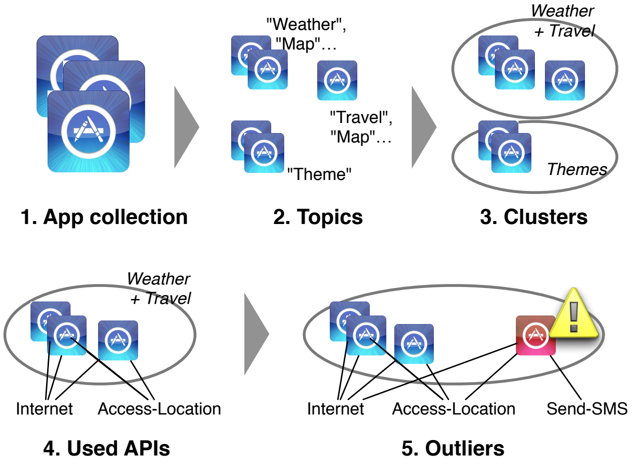


Figure 1: Detecting applications with unadvertised behavior. Starting from a collection of “good” apps (1), we identify their description topics (2) to form clusters of related apps (3). For each cluster, we identify the sentitive APIs used (4), and can then identify outliers that use APIs that are uncommon for that cluster (5).

- An app that sends a text message to a premium number to raise money is suspicious? Maybe, but on Android, this is a legitimate payment method for unlocking game features.
- An app that tracks your current position is malicious? Not if it is a navigation app, a trail tracker, or a map application.
- An application that takes all of your contacts and sends them to some server is malicious? This is what *WhatsApp* does upon initialization, one of the world’s most popular mobile messaging applications.

The question thus is not whether the behavior of an app matches a specific pattern or not; it is *whether the program behaves as advertised.* In all the examples above, the user would be informed and asked for authorization before any questionable behavior. It is the *covert* behavior that is questionable or downright malicious.

In this paper, we attempt to check *implemented* app behavior against *advertised* app behavior. Our domain is Android apps, so chosen because of its market share and history of attacks and frauds. As a proxy for the advertised behavior of an app, we use its natural language description from the Google Play Store. As a proxy for its implemented behavior, we use the set of Android *application programming interfaces* (APIs) that are used from within

[*] Ilaria Tavecchia is now with SWIFT, Brussels, Belgium.

ICSE ’14, May 31 - June 7, 2014, Hyderabad, India

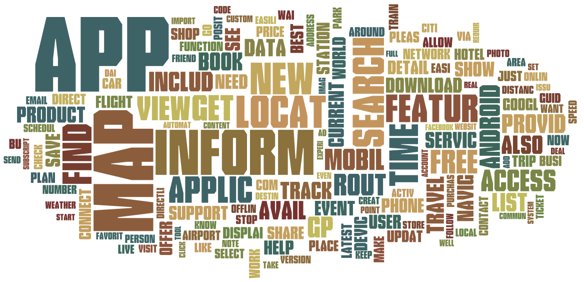


Figure 2: Words in app descriptions of the "navigation and travel" cluster. The bigger a word, the more app descriptions it appears in.

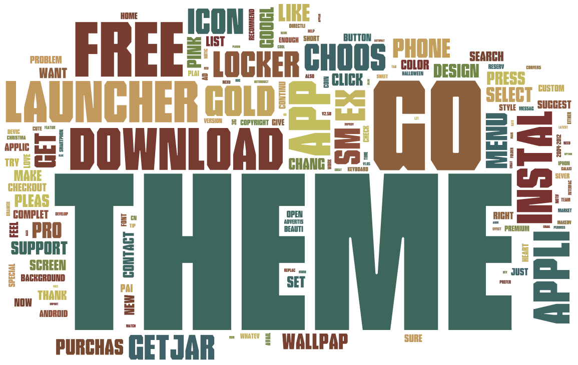


Figure 3: Words in "personalize" apps

the app binary. The key idea is to associate descriptions and API usage to detect anomalies: "This 'weather' application accesses the messaging API, which is unusual for this category."

Specifically, our CHABADA approach[1] takes five steps, illustrated in Figure 1 and detailed later in the paper:

1. CHABADA starts with a *collection* of 22,500+ "good" Android applications downloaded from the Google Play Store.

2. Using Latent Dirichlet Allocation (LDA) on the app descriptions, CHABADA identifies the main *topics* ("theme", "map", "weather", "download") for each application.

3. CHABADA then *clusters* applications by related topics. For instance, apps related to "navigation" and "travel" share several topics in their description, so they form one cluster.

   Figure 2 shows the word cloud for the descriptions of the apps in the "navigation and travel" cluster. Clusters very much differ by their word clouds; compare, for instance, the word cloud for the "personalization" cluster (Figure 3).

4. In each cluster, CHABADA identifies the APIs each app statically accesses. We only consider *sensitive* APIs, which are governed by a *user permission*. For instance, APIs related to internet access are controlled by an "INTERNET" permission.

   Figure 4 shows the sensitive APIs used by "navigation and travel" applications, grouped by the governing permission. Obviously, the "normal" behavior for these apps is to access the current location and access the internet (typically, a map server). Internet access is also used by "personalize" apps (Figure 5); however, they would access external storage and components rather than the current location.

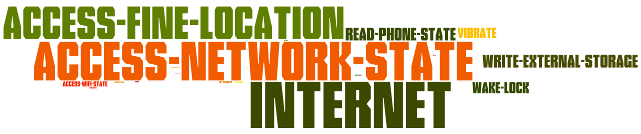


Figure 4: APIs, grouped by associated permission, used within the "navigation and travel" cluster. Standard behavior is to access the location ("ACCESS-FINE-LOCATION") and access the Internet ("ACCESS-NETWORK-STATE", "INTERNET").

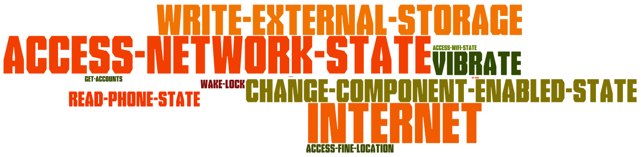


Figure 5: APIs for the "personalize" cluster. In contrast to Figure 4, these apps frequently access the SD card ("WRITE-EXTERNAL-STORAGE"), enable and disable new features ("CHANGE-COMPONENT-ENABLED-STATE"), but rarely, if ever, access the device location.

5. Using unsupervised One-Class SVM anomaly classification, CHABADA identifies *outliers* with respect to API usage. It produces a *ranked list of applications* for each cluster, where the top apps are most abnormal with respect to their API usage—indicating possible mismatches between description and implementation. Likewise, yet unknown applications would first be assigned to the cluster implied by their description and then be *classified* as being normal or abnormal.

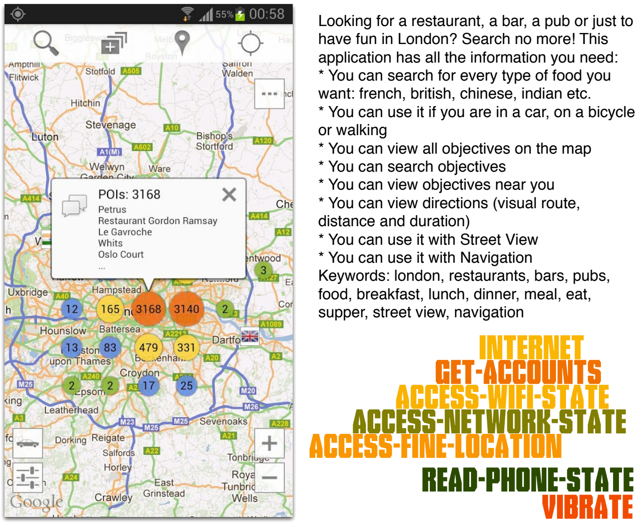
   In the "navigation and travel" cluster, the usage of any API that changes the configuration of the phone or its components would be unusual; this, however, is common for "personalize" apps, which in turn rarely access the current location. In both clusters, any app that would read or write contacts, calendar entries, pictures, text messages, the browser history, network settings, etc. would immediately be flagged as an outlier—and thus be subject to further scrutiny.

By flagging anomalous API usage within each cluster, CHABADA is set up to detect any kind of *mismatch between advertised and implemented behavior.* How does this work in practice? Figure 6 shows *London Restaurants & Pubs +*[2], available from the Google Play Store. Its description clearly puts it into the "navigation and travel" cluster. Besides expected API calls to access the current location and the internet, however, it also uses three API calls to retrieve the list of user accounts on the device—*getAccountsByType()*, *getDeviceId()*, and *getLine1Number()*, all governed by the "GET-ACCOUNTS" permission. These calls, which retrieve sensitive information such as the device identifier and mobile phone number, make *London Restaurants* an outlier within "navigation and travel".

Indeed, *London Restaurants* covertly sends this information (including additional information such as the current location) to its advertisement service, which is not mentioned at all in the description.

[1] CHABADA stands for CHecking App Behavior Against Descriptions of Apps. "Chabada" is a French word for the base ternary rhythm pattern in Jazz.

[2] https://play.google.com/store/apps/details?id=com.alarisstudio.maps.restaurants.london



Looking for a restaurant, a bar, a pub or just to have fun in London? Search no more! This application has all the information you need:
* You can search for every type of food you want: french, british, chinese, indian etc.
* You can use it if you are in a car, on a bicycle or walking
* You can view all objectives on the map
* You can search objectives
* You can view objectives near you
* You can view directions (visual route, distance and duration)
* You can use it with Street View
* You can use it with Navigation
Keywords: london, restaurants, bars, pubs, food, breakfast, lunch, dinner, meal, eat, supper, street view, navigation

Figure 6: The app *London Restaurants Bars & Pubs +*, together with complete description and API groups accessed

Is this malware? Possibly. Is this unexpected behavior? Certainly.[3] If *London Restaurants* had been explicit about what it does, it would have fallen in an "advertisements" cluster instead, where it would no longer be an outlier.

In our research, we found several more examples of false advertising, plain fraud, masquerading, and other questionable behavior. As a side effect, our approach is also effective as a malware detector: Training per-cluster SVM classifiers on benign applications, CHABADA flagged 56% of known malware as such, without requiring any training on malware patterns.

The remainder of this paper is organized as follows. We first detail how to cluster applications by description topics in Section 2. Section 3 describes how in each cluster we detect outliers with respect to their API usage. Section 4 evaluates our approach, manually and automatically, quantitatively and qualitatively. After discussing the related work (Section 5), Section 6 closes with conclusion and consequences.

2. CLUSTERING APPS BY DESCRIPTION

The intuition behind CHABADA is simple: applications that are similar, in terms of their descriptions, should also behave similarly. For this, we must first establish what makes two descriptions "similar". We start with our collection method for Android apps (Section 2.1). After initial processing (Section 2.2), CHABADA identifies *topics* of app descriptions (Section 2.3), and then *clusters* the apps based on common topics (Section 2.4 to Section 2.6).

2.1 Collecting Applications

Our approach is based on detecting anomalies from "normal", hopefully benign applications. As a base for such "normal" behavior, we collected a large set of applications from the Google Play Store, the central resource for Android apps. Our automated collection script ran at regular intervals during the Winter and Spring of 2013, and for each of the 30 categories in the Google Play Store, downloaded the top 150 free[4] applications in each category. A single complete run of our script thus returned 4,500 apps; as the top 150 apps shifted during our collection, we obtained a total of 32,136 apps across all categories.

In addition to the actual app (coming as an APK file), we also collected the store *metadata*—such as name, description, release date, user ratings, or screenshots. As CHABADA is set to identify outliers before they get released to the public, it only uses name and description.

2.2 Preprocessing Descriptions with NLP

Before subjecting our descriptions to topic analysis, we applied standard techniques of natural language processing (NLP) for filtering and stemming.

App descriptions in the Google Play Store frequently contain paragraphs in multiple languages—for instance, the main description is in English, while at the end of the description developers add a short sentence in different languages to briefly describe the application. To be able to cluster similar descriptions, we had to choose one single language, and because of its predominance we chose English. To remove all paragraphs of text that were not in English, we ran Google's *Compact Language Detector*[5] to detect their most likely language; non-English paragraphs were removed.

After multi-language filtering, we removed *stop words* (common words such as "the", "is", "at", "which", "on", . . .), and applied *stemming* on all descriptions. Stemming is a common NLP technique to identify the word's root, and it is essential to make words such as "playing", "player", and "play" all match to the single common root "plai". Stemming can improve the results of later NLP processes, since it reduces the number of words. We also removed non-text items such as numerals, HTML tags, links and email addresses.

As an example, consider the description of *London Restaurants* in Figure 6; after stop word removal and stemming, it appears as:

```
look restaur bar pub just fun london search applic inform need
can search everi type food want french british chines indian etc
can us car bicycl walk can view object map can search object
can view object near can view direct visual rout distanc durat
can us street view can us navig keyword london restaur bar pub
food breakfast lunch dinner meal eat supper street view navig
```

With that, we eliminated those applications from our set whose description would have less than 10 words after the above NLP preprocessing. Also, we eliminated all applications without any sensitive APIs (see Section 3 for details). This resulted in a final set of 22,521 apps, which form the base for our approach.

2.3 Identifying Topics with LDA

To identify sets of topics for the apps under analysis, we resort to *topic modeling* using *Latent Dirichlet Allocation* (LDA) [4].

LDA relies on statistical models to discover the topics that occur in a collection of unlabeled text. A "topic" consists of a cluster of words that frequently occur together. By analyzing a set of app descriptions on navigation and travels, for instance, LDA would group words such as "map", "traffic", "route", and "position" into one cluster, and "city", "attraction", "tour", and "visit" into another cluster. Applications whose description is mainly about navigation would thus be assigned to the first topic, since most of the words occurring in the description belong to the first cluster. Applications such as *London Restaurants*, however, would be assigned to both topics, as the words in the description appear in both clusters.

Our implementation feeds output of NLP pre-processing (i.e., the English text without stop words, and after stemming) into the *Mallet* framework [18]. We could freely choose the number of topics to

[3] When installing *London Restaurants*, the user must explicitly acknowledge its set of permissions, but why would the user find something like "account access" unusual or suspicious?

[4] Section 4.3 discusses possible induced bias.

[5] http://code.google.com/p/chromium-compact-language-detector/

Table 1: Topics mined from Android Apps

Id	Assigned Name	Most Representative Words (stemmed)
0	"personalize"	galaxi, nexu, device, screen, effect, instal, customis
1	"game and cheat sheets"	game, video, page, cheat, link, tip, trick
2	"money"	slot, machine, money, poker, currenc, market, trade, stock, casino coin, finance
3	"tv"	tv, channel, countri, live, watch, germani, nation, bbc, newspap
4	"music"	music, song, radio, play, player, listen
5	"holidays" and religion	christmas, halloween, santa, year, holiday, islam, god
6	"navigation and travel"	map, inform, track, gps, navig, travel
7	"language"	language, word, english, learn, german, translat
8	"share"	email, ad, support, facebook, share, twitter, rate, suggest
9	"weather and stars"	weather, forecast, locate, temperatur, map, city, light
10	"files and video"	file, download, video, media, support, manage, share, view, search
11	"photo and social"	photo, friend, facebook, share, love, twitter, pictur, chat, messag, galleri, hot, send social
12	"cars"	car, race, speed, drive, vehicl, bike, track
13	"design and art"	life, peopl, natur, form, feel, learn, art, design, uniqu, effect, modern
14	"food and recipes"	recip, cake, chicken, cook, food
15	"personalize"	theme, launcher, download, install, icon, menu
16	"health"	weight, bodi, exercise, diet, workout, medic
17	"travel"	citi, guid, map, travel, flag, countri, attract
18	"kids and bodies"	kid, anim, color, girl, babi, pictur, fun, draw, design, learn
19	"ringtones and sound"	sound, rington, alarm, notif, music
20	"game"	game, plai, graphic, fun, jump, level, ball, 3d, score
21	"search and browse"	search, icon, delet, bookmark, link, homepag, shortcut, browser
22	"battle games"	story, game, monster, zombi, war, battle
23	"settings and utils"	screen, set, widget, phone, batteri
24	"sports"	team, football, leagu, player, sport, basketbal
25	"wallpapers"	wallpap, live, home, screen, background, menu
26	"connection"	device, connect, network, wifi, blootooth, internet, remot, server
27	"policies and ads"	live, ad, home, applovin, notif, data, polici, privacy, share, airpush, advertis
28	"popular media"	seri, video, film, album, movi, music, award, star, fan, show, gangnam, top, bieber
29	"puzzle and card games"	game, plai, level, puzzl, player, score, challeng, card

be identified by LDA; and we chose 30, the number of categories covered by our apps in the Google Play Store. Furthermore, we set up the LDA such that an app would belong to at most 4 topics, and we consider an app related to a topic only if its probability for that topic is at least 5%.

Table 1 shows the resulting list of topics for the 22,521 descriptions that we analyzed; the "assigned name" is the abstract concept we assigned to that topic. Our example application, *London Restaurants*, is assigned to three of these topics:

- Topic 6 ("navigation and travel") with a probability of 59.8%,
- Topic 14 ("food and recipes") with a probability of 19.9%, and
- Topic 17 ("travel") with a probability of 14.0%.

2.4 Clustering Apps with K-means

Topic modeling assigns an application description to each topic with a certain probability. In other words, each application is characterized by a vector of affinity values (probabilities) for each topic. However, what we want is to identify *groups* of applications with similar descriptions, and we do that using the K-means algorithm, one of the most common clustering algorithms [16].

Given a set of elements in a metric space, and the number K of desired clusters, K-means selects one *centroid* for each cluster, and then associates each element of the data set with the nearest centroid, thus identifying clusters. In this context, the elements to be clustered are the applications as identified by their vector of affinities to topics.

Table 2 shows four applications $app_1, \ldots, app_4$, with the corresponding probabilities of belonging to four topics. When applied to this set of applications with $K = 2$ clusters, K-means returns one cluster with app_1 and app_3, and another cluster with app_2 and app_4.

Table 2: Four applications and their likelihoods of belonging to specific topics

Application	$topic_1$	$topic_2$	$topic_3$	$topic_4$
app_1	0.60	0.40	—	—
app_2	—	—	0.70	0.30
app_3	0.50	0.30	—	0.20
app_4	—	—	0.40	0.60

2.5 Finding the Best Number of Clusters

One of the challenges with K-means is to estimate the number of clusters that should be created. The algorithm needs to be given either some initial potential centroids, or the number K of clusters to identify. There exist several approaches to identify the best solution, among a set of possible solutions. Therefore, we run K-means several times, each time with a different K number, to obtain a set of clusterings we would then be able to evaluate. The range for K covers solutions among two extremes: having a small number of clusters (even just 2) with a large variety of apps; or having many clusters (potentially even one per app) and thus being very specific. We fixed $num\_topics \times 4$ as an upper bound, since in our settings an application can belong to up to 4 topics.

To identify the best solution, i.e., the best number of clusters, we used the *elements silhouette*, as discussed in [21]. The silhouette of an element is the measure of how closely the element is matched to the other elements within its cluster, and how loosely it is matched to other elements of the neighboring clusters. When the value of the silhouette of an element is close to 1, it means that the element is in the appropriate cluster. If the value is close to -1, instead, it means that the element is in the wrong cluster. Thus, to identify the best solution, we compute the average of the elements' silhouette for each solution using K as the number of clusters, and we select the solution whose silhouette was closest to 1.

2.6 Resulting App Clusters

Table 3 shows the list of clusters that were identified for the 22,521 apps that we analyzed. Each of these 32 clusters contains apps whose descriptions contain similar topics, listed under "Most Important Topics". The percentages reported in the last column represent the weight of specific topics within each cluster.

The clusters we identified are quite different from the *categories* one would find in an app store such as the Google Play Store. Cluster 22 ("advertisements"), for instance, is filled with applications that do nothing but display ads in one way or another; these apps typically promise or provide some user benefit in return. Cluster 16 ("connection") represents all application that deal with Bluetooth, Wi-Fi, etc.; there is no such category in the Google Play Store. The several "wallpaper" clusters, from adult themes to religion, simply represent the fact that several apps offer very little functionality.

Table 3: Clusters of applications. "Size" is the number of applications in the respective cluster. "Most Important Topics" list the three most prevalent topics; most important (> 10%) shown in bold. Topics less than 1% not listed.

Id	Assigned Name	Size	Most Important Topics
1	"sharing"	1,453	**share** (53%), settings and utils, navigation and travel
2	"puzzle and card games"	953	**puzzle and card games** (78%), share, game
3	"memory puzzles"	1,069	**puzzle and card games** (40%), game (12%), share
4	"music"	714	**music** (58%), share, settings and utils
5	"music videos"	773	**popular media** (44%), **holidays and religion** (20%), share
6	"religious wallpapers"	367	**holidays and religion** (56%), design and art, wallpapers
7	"language"	602	**language** (67%), share, settings and utils
8	"cheat sheets"	785	**game and cheat sheets** (76%), share, popular media
9	"utils"	1,300	**settings and utils** (62%), share, connection
10	"sports game"	1,306	**game** (63%), battle games, puzzle and card games
11	"battle games"	953	**battle games** (60%), **game** (11%), design and art
12	"navigation and travel"	1,273	**navigation and travel** (64%), share, travel
13	"money"	589	**money** (57%), puzzle and card games, settings and utils
14	"kids"	1,001	**kids and bodies** (62%), share, puzzle and card games
15	"personalize"	304	**personalize** (71%), **wallpapers** (15%), settings and utils
16	"connection"	823	**connection** (63%), settings and utils, share
17	"health"	669	**health** (63%), design and art, share
18	"weather"	282	**weather and stars** (61%), **settings and utils** (11%), navigation and travel
19	"sports"	580	**sports** (62%), share, popular media
20	"files and videos"	679	**files and videos** (63%), share, settings and utils
21	"search and browse"	363	**search and browse** (64%), game, puzzle and card games
22	"advertisements"	380	**policies and ads** (97%)
23	"design and art"	978	**design and art** (48%), share, game
24	"car games"	449	**cars** (51%), game, puzzle and card games
25	"tv live"	500	**tv** (57%), share, navigation and travel
26	"adult photo"	828	**photo and social** (59%), share, settings and utils
27	"adult wallpapers"	543	**wallpapers** (51%), share, kids and bodies
28	"ad wallpapers"	180	**policies and ads** (46%), wallpapers, settings and utils
29	"ringtones and sound"	662	**ringtones and sound** (68%), share, settings and utils
30	"theme wallpapers"	593	**wallpapers** (90%), holidays and religion, share
31	"personalize"	402	**personalize** (86%), share, settings and utils
32	"settings and wallpapers"	251	**settings and utils** (37%), **wallpapers** (37%), personalize

The *London Restaurants* app ended up in Cluster 12, together with other applications that are mostly about navigation and travels. These are the clusters of apps related by their descriptions in which we now can search for outliers with respect to their behavior.

2.7 Alternative Clustering Approaches

As with most scientific work, the approach presented in this paper only came to be through several detours, dead-ends, and refinements. We briefly list the most important ones here as to have future researchers avoid some of the problems we encountered.

Usage of topics. One might wonder if it is really necessary to cluster based on topics instead of clustering plain descriptions directly. The reason is that K-means, as well as any other clustering algorithm, works better when few features are involved. Hence, abstracting descriptions into topics was crucial to obtain better clustering results.

Usage of clusters. Having just one dominant topic for applications did not yield better results, since several applications may incorporate multiple topics at once. This also excluded the usage of the given Google Play Store categories as a clustering strategy. Despite one might argue that clustering does not produce different results than just clustering on the predominant topics (the number of topics and cluster is almost the same), one should also notice that clusters have quite different features than topics. For instance, Cluster 22 ("advertisements") groups applications whose main topic is about wallpapers and mention in the description that the application is using advertisements. This contrasts to Cluster 32 ("settings and wallpapers"), for instance, which also groups applications that are about wallpapers, but do *not* mention advertisements in the description.

One cluster per app. As it is now, each application belongs to one cluster, which may incorporate multiple topics. This leads to a good clustering of similar apps. A yet unexplored alternative is to allow an app to be a member of *multiple clusters.* This might potentially provide better clustering results.

Choice of clustering method. Before using K-means, we experimented with *formal concept analysis* to detect related concepts of topics and features [25] without successful results; the analysis was overwhelmed by the number of apps and features. K-means has known limitations, and we believe that other clustering algorithms could improve the clustering.

Low quality apps. App stores like the Google Play Store contain several free applications of questionable value. Restricting our approach to a minimum number of downloads or user ratings may yield very different results. However, the goal of our approach is to identify outliers before users see them, and consequently we should consider all apps.

3. IDENTIFYING OUTLIERS BY APIS

Now that we have clustered apps based on similarity of their description topics, we can search for outliers regarding their actual behavior. Section 3.1 shows how we extract API features from Android binaries. Section 3.2 focuses on APIs controlled by permissions. Section 3.3 describes how CHABADA detects API outliers.

3.1 Extracting API Usage

As discussed in the introduction, we use *static API usage* as a proxy for behavior. Going for API usage is straightforward: while Android bytecode can also be subject to advanced static analysis such as information flow analysis and standard obfuscation techniques that easily thwart any static analysis, API usage has to be explicitly declared; in Android binaries, as in most binaries on other platforms, static API usage is easy to extract. For each Android application, we extracted the (binary) APK file with *apktool*[6], and with a *smali* disassembler, we extracted all API invocations, including the *number of call sites* for each API.

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

Table 4: Sensitive APIs used in *London Restaurants*. The bold APIs make this app an outlier in its cluster.

android.net.ConnectivityManager.getActiveNetworkInfo()
android.webkit.WebView()
java.net.HttpURLConnection.connect()
android.app.NotificationManager.notify()
java.net.URL.openConnection()
android.telephony.TelephonyManager.getDeviceId()
org.apache.http.impl.client.DefaultHttpClient()
org.apache.http.impl.client.DefaultHttpClient.execute()
android.location.LocationManager.getBestProvider()
android.telephony.TelephonyManager.getLine1Number()
android.net.wifi.WifiManager.isWifiEnabled()
android.accounts.AccountManager.getAccountsByType()
android.net.wifi.WifiManager.getConnectionInfo()
android.location.LocationManager.getLastKnownLocation()
android.location.LocationManager.isProviderEnabled()
android.location.LocationManager.requestLocationUpdates()
android.net.NetworkInfo.isConnectedOrConnecting()
android.net.ConnectivityManager.getAllNetworkInfo()

3.2 Sensitive APIs

Using *all* API calls as features would induce overfitting in later stages. Therefore, we select a subset of APIs, namely the *sensitive* APIs that are governed by an Android *permission setting.* These APIs access sensitive information (such as the user's picture library, the camera, or the microphone) or perform sensitive tasks (altering system settings, sending messages, etc.) When installing an app, the user must explicitly *permit* usage of these APIs. For this purpose, each Android app includes a manifest file which lists the permissions that the application requires for its execution. To obtain the set of sensitive APIs, we relied on the work of Felt et al., who identified and used the mapping between permissions and Android methods [7]; we considered a sensitive API to be used by the app if and only if it is declared in the binary and if its corresponding permission is requested in the manifest file. This allowed us to eliminate API calls that are used within third party libraries, and not used by the application directly.

As an example for such sensitive APIs, consider Table 4. These are the APIs used by the *London Restaurants* app that would be governed by a specific permission. Through these APIs, the app accesses the current network provider, the last known location (as well as updates), and the internet via an HTTP connection. These APIs also reveal that the app accesses the device identifier, the user's account info, as well as the mobile phone "line1" number. These latter APIs, shown in bold, would be governed by the "GET-ACCOUNTS" permission. As each permission governs several APIs, using permissions alone would give us too few features to learn from. Instead, the sensitive APIs allow for a much more fine-grained characterization of the application behavior.

3.3 Identifying API Outliers with OC-SVMs

Now that we have all API features for all apps, the next step is to identify *outliers*—that is, those applications whose API usage would be abnormal within their respective topic cluster. To identify these outliers, we use *One-Class Support Vector Machine learning* (OC-SVM) [22], which is a machine learning technique to learn the features of *one class* of elements. The resulting SVM model can later be used for anomaly/novelty detection within this class. (Note how this is in contrast to the more common usage of Support Vector Machines as classifiers, where each app additionally has to be *labeled* as belonging to a specific class—say, "benign" vs. "malicious"—during training.)

OC-SVM has been successfully applied in various contexts that span from document classification [17] to automatic detection of anomalous Windows registry accesses [10]. In our context, the interesting feature of OC-SVM is that one can provide only samples of one class (say, of regular benign applications), and the classifier will be able to identify samples belonging to the same class, tagging the others as anomalies. OC-SVMs, therefore, are mainly used in those cases in which there exist many samples of one class of elements (e.g. benign applications), and not many samples of other classes (e.g. malicious applications).

With the sensitive APIs as binary features, CHABADA trains an OC-SVM within each cluster with a subset of the applications in order to model which APIs are commonly used by the applications in that cluster. The resulting cluster-specific models are then used to identify the outlier applications, i.e., applications whose used APIs differ from the common use of the API within the same cluster.

To represent the distance of an element from the common behavior, we use the actual distance of the element from the hyperplane that the OC-SVM builds. The bigger this distance, the further an element is from the commonly observed behavior. Thus, by ranking the elements (i.e. apps) by their distance to the OC-SVM hyperplane, we can identify which ones have behaviors that differ the most from what has been commonly observed.

As an example, consider again our *London Restaurants* app. After training the OC-SVM on the apps in Cluster 12, it classifies *London Restaurant* as an outlier. The reason is the APIs shown in bold in Table 4—indeed, accessing the device identifier, the user's account info, or his mobile phone number is uncommon for the apps in the "navigation and travel" cluster. In our evaluation in Section 4, we discuss trends that make an app an outlier.

3.4 Alternative Approaches

To detect anomalies, we have identified several alternative settings; some of which we already experimented with. Again, we briefly list them here.

Class and package names as abstraction. Before going for sensitive APIs with Felt et al.'s mapping, we considered abstracting the API method invocations by considering only the class name or the package name instead of the whole method signature. Although this helped reducing the number of features, it also caused a loss of relevant information. Invocations to sensitive methods such as *getLine1Number()*, which returns the user's phone number, would be indistinguishable from relatively harmless methods such as *getNetworkType()*, which returns the current data connection, since they are both declared in class *TelephonyManager*.

Number of call sites. For each API we considered a binary value (i.e. there exists at least one call site for that API in the app or not). We could have considered the normalized number of call sites for each API as a feature. We expect similar results, although we only experimented with binary values.

TF-IDF for APIs. Since some APIs are commonly used across all clusters, for instance APIs for Internet access, we could have considered only the most relevant and representative methods for the cluster instead of all the methods. *Term frequency-inverse document frequency* (TF-IDF) [13] could filter out some of the non discriminant features, thus providing even greater support for the OC-SVM algorithm. We have not tried this path yet, although we plan to investigate it in the near future.

Insensitive APIs. To avoid overfitting, limiting the number of features is crucial. Hence, we focused on APIs that would be "sensitive", that is, impacted by Android permissions. Expanding this set to other relevant APIs might yield even better results.

Permissions instead of APIs. Instead of APIs, we could have used the list of permissions in the manifest file as features. However, studies showed that almost 30% of Android applications request more permissions than they actually use [7, 2, 3, 23]. We chose APIs since they provide a more fine-grained view into what apps do and do not.

Avoiding APIs as predictors alone. When training a classifier on descriptions and APIs of known malware, the specific APIs being used in the malware set (typically, sending text messages) will dominate the descriptions. By first clustering by descriptions and then classifying, we obtain better results.

4. EVALUATION

To evaluate the effectiveness of our technique, we investigated the following main research questions:

RQ1 *Can our technique effectively identify anomalies (i.e., mismatches between description and behavior) in Android applications?* For this purpose, we manually inspected the top outliers as produced by our technique and classified them with respect to covert behavior (Section 4.1).

RQ2 *Can our technique be used to identify malicious Android applications?* For this purpose, we included in our set of applications a set of known malware, and we ran OC-SVM as a classifier (Section 4.2).

4.1 Outlier Detection

Let us start with RQ1: *Can our technique effectively identify anomalies (i.e., mismatches between description and behavior) in Android applications?* For this purpose, we ran CHABADA on all 32 clusters, as described in Section 3. Following K-fold validation, we partitioned the entire set of 22,521 applications in 10 subsets, and we used 9 subsets for training the model and 1 for testing. We ran this 10 times, each time considering a different subset for testing. Out of the whole list of outliers identified in each run, we identified the top 5 outliers from the ranked list for each cluster. These 160 outliers would now have to be assessed whether they would really exhibit suspicious behavior.

4.1.1 Manual Assessment

In the end, only a human can interpret properly what is in an app description. Therefore, for these 160 applications, we would manually examine their description, the list of APIs used, as manually inspect the code. We would classify each of the apps into one of three categories:

Malicious – the app shows unadvertised (covert) behavior using sensitive APIs that acts against the interest of its users.

Dubious – the app shows unadvertised (covert) behavior using sensitive APIs, but would *not* necessarily act against the user's interests.

Benign – all sensitive behavior is properly described. This includes apps which clearly list the sensitive data they collect, and also applications placed in the wrong cluster due to inadequate descriptions.

We applied an "innocent unless proven guilty" principle: On average, each such assessment would take 2 minutes for benign applications, and up to 5 minutes for dubious or malicious applications.

Table 5 summarizes our assessment. Overall, we clearly identified 42 outliers as malware, which is 26% of our sample. Given that these apps stem from an app store that is supposed to maintain quality, this is a worrying result.[7] Even more worrying is that there are clusters of apps in which the majority of outliers are all malicious. In Clusters 14 ("kids"), 16 ("connection"), 28 ("ad wallpapers"), 30 ("theme wallpapers"), the majority of outliers are malicious.

The good news in all of this is that the outliers as reported by CHABADA are worth looking into: 39% of the top 5 outliers require additional scrutiny by app store managers—or end users, who may just as well run CHABADA for their protection.

Top outliers, as produced by CHABADA, contain 26% malware; additional 13% apps show dubious behavior.

4.1.2 What Makes an Outlier?

During our investigation, we identified a number of repeating trends that determined whether an app would be an outlier or not. These trends can be characterized as follows:

Spyware ad frameworks.

A large portion of the "malicious" apps we found are spyware. We identified multiple applications in different clusters that get sensitive information such as the user's phone number, the device id, the user's current location, and the list of emails used in different accounts such as the Google and Facebook accounts. Apps do not use this information for themselves, but they retrieve these data only because they include third party libraries for advertisements, and advertisement companies such as *apploving* and *airpush* pay application developers for users' sensitive information. Some apps clearly state that they include such frameworks, and most of them ended up in the advertisements cluster, where such behavior is normal. Apps that do not mention the usage of such frameworks and their impact on privacy, however, are spyware.

Just to mention a few examples, we found "mosquito killer" apps such as *Mosquito Repellent – No Ads*, *Anti-Mosquitoes*, *Mosquito Repellent Plus*, wedding apps such as *Wedding Ideas Gallery*, and apps with collections of wallpapers such as *Christmas girl live wallpaper* and *Twilight live wallpaper*. Since they all include the same ad frameworks, they all get the sensitive data mentioned above and send it to ad servers.

Dubious behavior.

In "dubious" applications whose description does not completely justify the behavior. For instance, the *UNO Free* game application accesses the user's location without explanation, and *WICKED*, the official application for a Broadway show, can record audio, but it does not say for which purposes. *Runtastic*, a widely used training application, can also record audio, but it does not mention it in the description. Finally, *Yahoo! Mail*, which is the official application to browse and compose emails with Yahoo, can send SMSs. From the description it is not clear why the application should do that.

Misclassified apps.

Some "benign" applications were misclassified on the basis of their descriptions, and consequently were assigned to clusters popu-

[7] Note, though, that between the time of download and the time of writing, many of these applications had already been identified as malicious by users and since been removed.

Table 5: Manual assessment of the top 5 outliers, per cluster and total.

Behavior	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	Total
Malicious	1	2	1	0	0	0	0	2	0	0	0	0	0	3	0	4	2	0	0	1	3	5	1	3	1	1	2	3	1	4	1	1	**42** (26%)
Dubious	1	2	1	0	0	0	0	0	0	2	2	1	1	0	0	0	1	2	1	1	0	0	2	0	1	0	1	0	1	0	0	0	**20** (13%)
Benign	3	1	3	5	5	5	5	3	5	3	3	4	4	2	5	1	2	3	4	3	2	0	2	2	3	4	2	2	3	1	4	4	**98** (61%)

lated by applications that cover substantially different topics. For instance, *SlideIT free Keyboard*, which is a popular tool to insert text by sliding the finger along the keyboard letters, ended up in the language cluster, since more than half of the description is about the language support. *Romanian Racing*, which is a racing game, also ended up in the same cluster, mainly because its brief description mentions multi-language support.

We also had some rare cases of applications that were associated to completely unexpected clusters. *Hamster Life*, a pet training game, ended up in the "religious wallpapers" cluster.

Uncommon behavior.

Some apps were tagged as outliers because their behavior, although benign and clearly described, is just very uncommon for the clusters these applications belong to. For instance, *SoundCloud – Music & Audio* was tagged as an anomaly mainly because it records audio. This is expected behavior, since the application connects to a platform for finding and sharing new music, and it allows to record music that users produce. However, recording audio is not one of the common features of Cluster 1, and consequently *SoundCloud* was tagged as an anomaly. Similarly, *Llama – Location Profiles*, which allows to change the ringtones depending on the context and location, was tagged as an anomaly because it is not common for personalization applications to access the user's location and calendar. This application, though, uses this information to automatically switch between vibrate and ringtones when the user, for instance, is at home, in office, or has a meeting.

Benign outliers.

In some clusters, CHABADA identified the uncommon behavior as the *lack of malicious behavior*. For instance, Cluster 13 ("money") contains several applications that use libraries for advertisements, and thus access sensitive information. As a consequence, *Mr. Will's Stud Poker* and *Mr. Will's Draw Poker* were tagged as anomalies because they do *not* access sensitive information, despite them being poker games.

> *CHABADA is behavior-agnostic: It cannot determine whether an application is good or bad, only if it is common or not.*

4.2 Malware Detection

Let us now turn to RQ2: *Can our technique be used to identify malicious Android applications?* For this purpose, we used the dataset of Zhou et al. [28] containing more than 1,200 known malicious apps for Android. In their raw form, these apps lack metadata such as title or description. As many of these apps are repackaged versions of an original app, we were able to collect the appropriate description from the Google Play Store. We used the title of the application and the package identifier to search for the right match in the Store. For 72 cases we could find exactly the same package identifier, and for 116 applications we found applications whose package identifier was very similar. We manually checked that the match was correct. As with our original set of "benign" apps (Section 2.1), we only kept those applications with an English description in the set, reducing it to 172 apps.

As a malware detector, we again used the OC-SVM model; but this time, we would use it as a *classifier*—that is, we used the SVM model for a *binary decision* on whether an element would be part of the same distribution or not.

4.2.1 Classification using topic clusters

We ran the classification on the "benign" set of apps used in the previous study, and we included the 172 malware samples for which we could find the corresponding description. We trained the OC-SVM only on "benign" applications as before, and we excluded the applications manually classified as "malicious" during the previous experiment (Section 4.1); We then trained within each cluster the OC-SVM on the APIs of 90% of these apps, and then used the OC-SVM as a classifier on a *testing set* composed of the known malicious apps in that cluster as well as the remaining 10% benign apps. What we thus simulated is a situation in which the *malware attack is entirely novel*—CHABADA must correctly identify the malware as such without knowing previous malware patterns.

As for the previous experiment, we would do this 10 times, each time considering a different test set. The number of malicious applications would *not* be equally distributed across clusters, as malicious applications are assigned to clusters depending on their descriptions. In our evaluation setting, with our data set, the number of malicious applications per cluster spans from 0 to 39.

The results of our classification are shown in Table 6. We report the average results of the 10 different runs. CHABADA correctly identifies 56% of the malicious apps as such, while only 16% of "benign" apps are misclassified. If our approach would be used to guard against yet unknown malicious behavior, it would detect the majority of malware as such.

Table 6: Checking APIs and descriptions within topic clusters (our approach)

	Predicted as malicious	Predicted as benign
Malicious apps	96.5 (56%)	75.5 (44%)
Benign apps	353.9 (16%)	1,884.4 (84%)

Compared against standard malware detectors, these results of course leave room for improvement—but that is because existing malware detectors compare against *known* malware, whose signatures and behavior are already known. For instance, accessing all user accounts on the device, as the *London Restaurants* app does, is a known pattern of malicious behavior. In practice, our approach would thus be used to *complement* such detectors, and be specifically targeted towards *novel* attacks which would be different from existing malware—but whose API usage is sufficiently abnormal to be flagged as an outlier.

> *In our sample, even without knowing existing malware patterns, CHABADA detects the majority of malware as such.*

4.2.2 Classification without clustering

We further evaluate the effectiveness of our approach by comparing it against alternatives. To show the impact of topic clustering, we compare our classification results against a setting in which the

OC-SVM would be trained on sensitive APIs and NLP-preprocessed words from the description alone—that is, all applications form one big cluster. As Table 7 shows, the malware detection rate decreases dramatically. This shows the benefits of our clustering approach.

Table 7: Checking APIs and descriptions in one single cluster

	Predicted as malicious	Predicted as benign
Malicious apps	41 (24%)	131 (76%)
Benign apps	334.9 (15%)	1,903.1 (85%)

Classifying without clustering yields more false negatives.

4.2.3 Classification using given categories

Finally, one may ask why our specific approach for clustering based on description topics would be needed, as one could also easily use the given store categories. To this end, we clustered the applications based on their categories in the Google Play Store, and repeated the experiment with the resulting 30 clusters. The results (Table 8) demonstrate the general benefits of clustering; however, topic clustering as in our approach, is still clearly superior. (Additionally, one may argue that a category is something that some librarian would assign, thus requiring more work and more data.)

Table 8: Checking APIs and descriptions within Google Play Store categories

	Predicted as malicious	Predicted as benign
Malicious apps	81.6 (47%)	90.4 (53%)
Benign apps	356.9 (16%)	1,881.1 (84%)

Clustering by description topics is superior to clustering by given categories.

4.3 Limitations and Threats to Validity

Like any empirical study, our evaluation is subject to threats to validity, many of which are induced by limitations of our approach. The most important threats and limitations are listed below.

External validity. CHABADA relies on establishing a relationship between description topics and program features from existing, assumed mostly benign, applications. We cannot claim that said relationships could be applied in other app ecosystems, or be transferable to these. We have documented our steps to allow easy replication of our approach.

Free apps only. Our sample of 22,521 apps is based on free applications only; i.e. applications that need to generate income through ads, purchases, or donations. Not considering paid applications makes our dataset biased. However, the bias would shift "normality" more towards apps supported by ads and other income methods, which are closer to undesired behavior exposed by malware. Our results thus are conservative and would rather be improved through a greater fraction of paid applications, which can be expected to be benign.

App and malware bias. Our sample also only reflects the top 150 downloads from each category in the Google Play Store. This sample is biased towards frequently used applications, and towards lesser used categories; likewise, our selection of malware (Section 4) may or may not be representative for current threats. Not knowing which actual apps are being used, and how, by Android users, these samples may be biased. Again, we allow for easy reproduction of our approach.

Researcher bias. Our evaluation of outliers is based on the classification by a single person, who is a co-author of this paper. This poses the risk of researcher bias, i.e. the desire of an author to come up with best possible results. To counter this threat, we are making our dataset publicly available (Section 6).

Native code and obfuscation. We limit our analyses to the Dalvik bytecode. We do not analyze native code. Hence, an application might rely on native code or use obfuscation to perform covert behavior; but then, such features may again characterize outliers; also, neither of these would change the set of APIs that must be called.

Static analysis. As we rely on static API usage, we suffer from limitations that are typical for static analysis. In particular, we may miss behavior induced through *reflection,* i.e. code generated at runtime. Although there exist techniques to statically analyze Java code using reflection, such techniques are not directly applicable with Android apps [5, 8]; in the long run, dynamic analysis paired with test generation may be a better option.

Static API declarations. Since we extract API calls statically, we may consider API calls that are never executed by the app. Checking statically whether an API is reached is an instance of the (undecidable) halting problem. As a workaround, we decided to consider an API only if the corresponding permission is also declared in the manifest.

Sensitive APIs. Our detection of sensitive APIs (Section 3.2) relies on the mapping by Felt et al. [7], which now, two years later, may be partially outdated. Incorrect or missing entries in the mapping would make CHABADA miss or misclassify relevant behavior of the app.

5. RELATED WORK

While this work may be the first to generally check app descriptions against app behavior, it builds on a history of previous work combining natural language processing and software development.

5.1 Mining App Descriptions

Most related to our work is the WHYPER framework of Pandita et al. [19]. Just like our approach, WHYPER attempts to automate the risk assessment of Android apps, and applies natural language processing to app descriptions. The aim of WHYPER is to tell whether the need for *sensitive permissions* (such as accesses to contacts or calendar) is motivated in the application description. In contrast to CHABADA, which fully automatically learns which topics are associated with which APIs (and by extension, which permissions), WHYPER requires manual annotation of sentences describing the need for permissions. Also, CHABADA goes beyond permissions in two ways: first, it focuses on APIs, which provide a more detailed view, and it aims for general mismatches between expectations and implementations.

The very idea of app store mining was introduced one year earlier when Harman et al. mined the Blackberry app store [9]. They focused on app meta-data to find patterns such as a correlation consumer rating and the rank of app downloads, but would not download or analyze the apps themselves.

Our characterization of "normal" behavior comes from mining related applications; in general, we assume what most applications in a well-maintained store do is also what most users would expect to be legitimate. In contrast, recent work by Lin et al. [14] suggests *crowdsourcing* to infer what users expect from specific privacy settings; just like we found, Lin et al. also highlight that privacy expectations vary between app categories. Such information from users can well complement what we infer from app descriptions.

5.2 Behavior/Description Mismatches

Our approach is also related to techniques that apply natural language processing to infer specifications from comments and documentation. Lin Tan et al. [24] extract implicit program rules from program corpora and use these rules to automatically detect inconsistencies between comments and source code, indicating either bugs or bad comments. Rules apply to ordering and nesting of calls and resource accesses ("f_a must not be called from f_b").

Høst and Østvold [11] learn from program corpora which verbs and phrases would normally be associated with specific method calls, and used these to identify misnamed methods.

Pandita et al. [20] identify sentences that describe code contracts from more than 2,500 sentences of API documents; these contracts can be checked either through tests or static analysis.

All these approaches compare program code against formal program documentation, whose semi-formal nature makes it easier to extract requirements. In contrast, CHABADA works on end-user documentation, which is decoupled from the program structure.

5.3 Detecting Malicious Apps

There is a large body of industrial products and research prototypes that focus on identifying known malicious behavior. Most influential for our work was the paper by Zhou and Jiang [28], who use the permissions requested by applications as a filter to identify potentially malicious applications; the actual detection uses static analysis to compare sequences of API calls against those of *known* malware. In contrast to all these approaches, CHABADA identifies outliers even without knowing what makes malicious behavior.

The TAINTDROID system [6] tracks dynamic information flow within Android apps and thus can detect usages of sensitive information. Using such dynamic flow information would yield far more precise behavior insights than static API usage; similarly, profilers such as ProfileDroid [26] would provide better information; however, both TAINTDROID and ProfileDroid require a representative set of executions. Integrating such techniques in CHABADA, combined with automated test generation [12, 27, 15, 1], would allow to learn normal and abnormal patterns of information flow; this is part of our future work (Section 6).

6. CONCLUSION AND CONSEQUENCES

By clustering apps by description topics, and identifying outliers by API usage within each cluster, our CHABADA approach effectively identifies applications whose behavior would be unexpected given their description. We have identified several examples of false and misleading advertising; and as a side effect, obtained a novel effective detector for yet unknown malware. Just like mining software archives has opened new opportunities for empirical software engineering, we see that mining apps and their descriptions opens several new opportunities for automated checking of natural-language requirements.

During our work, we have gained a number of insights into the Android app ecosystem that call for action. First and foremost, application vendors must be much more explicit about what their apps do to earn their income. App store suppliers such as Google should introduce better standards to avoid deceiving or incomplete advertising. Second, the way Android asks its users for permissions is broken. Regular users will not understand what "allow access to the device identifier" means, nor would they have means to check what is actually being done with their sensitive data, nor would they understand the consequences. Users understand, though, what regular apps do; and CHABADA is set to point out and highlight differences, which should be way easier to grasp.

Although our present approach came to be by exploring and refining several alternatives, we are well aware that it is by no means perfect or complete. Our future work will focus on the following topics:

Detailed Behavior patterns. Static API usage is a rather broad abstraction for characterizing what an app does and what not. More advanced methods could focus on the *interaction* of APIs, notably information flow between APIs.

Dynamic Behavior. Exploring actual executions would give a far more detailed view of what an app actually does—in particular, concrete values for all APIs accessing remote resources. We are working on GUI test generators for Android apps that aim for coverage of specific APIs or dynamic information flow.

Natural Language Processing. The state of the art in natural language processing can retrieve much more than just topics. Looking at dependencies between words (such as conjunctions, subject-verb, verb-object) could retrieve much more detailed patterns. Likewise, leveraging known ontologies would help in identifying synonyms.

A Rosetta Stone for Topics and Behavior. By mining thousands of applications, we can associate natural language descriptions with specific program behavior. The resulting mapping between natural language and program fragments help in program understanding as well as synthesis of programs and tests.

To allow easy reproduction and verification of our work, we have packaged all data used within this work for download. In particular, we have prepared a 50 MB dataset with the exact data that goes into CHABADA, including app names, descriptions, other metadata, permissions, and API usage. All of this can be found on the CHABADA web site:

`http://www.st.cs.uni-saarland.de/chabada/`

7. ACKNOWLEDGMENTS

We thank Vitalii Avdiienko, Juan Pablo Galeotti, Clemens Hammacher, Konrad Jamrozik, and Sascha Just for their helpful feedback on earlier revisions of this paper. Special thanks go to Andrea Fischer, Joerg Schad, Stefan Richter, and Stefan Schuh for their valuable assistance during the project.

This work was funded by the European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing".

8. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 258–261, New York, NY, USA, 2012. ACM.

[2] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *ACM Conference on Computer and Communications Security (CCS)*, pages 217–228, New York, NY, USA, 2012. ACM.

[3] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically securing permission-based software by reducing the attack surface: An application to Android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 274–277, 2012.

[4] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 241–250, New York, NY, USA, 2011. ACM.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[7] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security (CCS)*, pages 627–638, New York, NY, USA, 2011. ACM.

[8] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, 2013.

[9] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: MSR for app stores. In *IEEE Working Conference on Mining Software Repositories (MSR)*, pages 108–111, 2012.

[10] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo. One class support vector machines for detecting anomalous windows registry accesses. In *ICDM Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

[11] E. W. Høst and B. M. Østvold. Debugging method names. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 294–317. Springer, 2009.

[12] C. Hu and I. Neamtiu. Automating GUI testing for Android applications. In *International Workshop on Automation of Software Test (AST)*, pages 77–83, New York, NY, USA, 2011. ACM.

[13] K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[14] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *ACM Conference on Ubiquitous Computing (UbiComp)*, pages 501–510, New York, NY, USA, 2012. ACM.

[15] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 224–234, New York, NY, USA, 2013. ACM.

[16] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[17] L. M. Manevitz and M. Yousef. One-class SVMs for document classification. *Journal of Machine Learning Research*, 2:139–154, 2002.

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

[19] 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.

[20] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012.

[21] P. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, 1987.

[22] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001.

[23] R. Stevens, J. Ganz, P. Devanbu, H. Chen, and V. Filkov. Asking for (and about) permissions used by Android apps. In *IEEE Working Conference on Mining Software Repositories (MSR)*, pages 31–40, San Francisco, CA, 2013.

[24] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 145–158, 2007.

[25] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 35–44, New York, NY, 2007. ACM.

[26] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. ProfileDroid: multi-layer profiling of Android applications. In *ACM Annual International Conference on Mobile Computing and networking (MobiCom)*, pages 137–148, New York, NY, USA, 2012. ACM.

[27] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[28] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy (SP)*, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.