Detecting Behavior Anomalies in Graphical User Interfaces

Vitalii Avdiienko♣, Konstantin Kuznetsov♣, Isabelle Rommelfanger♣, Andreas Rau♣, Alessandra Gorla♠, Andreas Zeller♣
♣*Saarland University* ♠*IMDEA Software Institute*
Saarbrücken, Germany *Madrid, Spain*

***Abstract*—When interacting with user interfaces, do users always get what they expect? For each user interface element in thousands of Android apps, we extracted the Android APIs they invoke as well as the text shown on their screen. This association allows us to detect *outliers:* User interface elements whose text, context or icon suggests one action, but which actually are tied to other actions. In our evaluation of tens of thousands of UI elements, our BACKSTAGE prototype discovered misleading random UI elements with an accuracy of 73%.**

I. Introduction

One of the central principles in user interface design is *the principle of least astonishment*—that is, a user interface element should behave in a manner consistent with how its users expect it to behave. Such user expectations typically stem from *similar programs* which users are familiar with: If a UI element says "Print", "Save", "OK", "Close", or "Cancel", our experience with other programs using these labels gives us an idea of what to expect; and if the result does not match our expectation, we see this as a problem.

The possibly most dangerous mismatches, however, are those we do not even notice. Figure 1 shows the signup screen of TRIPWOLF, a popular travel guide app. To use its services, one has to sign up with a social network account or an e-mail address, clicking on "Join TRIPWOLF". The interesting thing about this Signup button is that it not only sends the e-mail address, but also the *precise user location* to the TRIPWOLF servers, using the *LocationManager* API. This is a problem, since Signup buttons normally is not supposed to send out our current location—and thus, this button shows a mismatch between user expectation and actual behavior.

In this paper, we check the *advertised* functionality of UI elements against their *implemented* functionality: "This Signup button should not send a location, but it does". The idea is to mine *app stores* – containing hundreds of thousands of apps with graphical user interfaces – and model users' expectations with respect to a particular UI element. The *expected behavior* results from similar UI elements in other apps.

Our approach consists of five steps, summarized in Figure 2:

App Collection We start with a collection of thousands of ANDROID apps, all taken from the Google Play Store by using the ANDROZOO [3] dataset.

Mining UI elements. We statically analyze the code and resources of each app to identify its set of UI elements, including those that would be set or changed dynamically. UI elements include text input, buttons, and more.

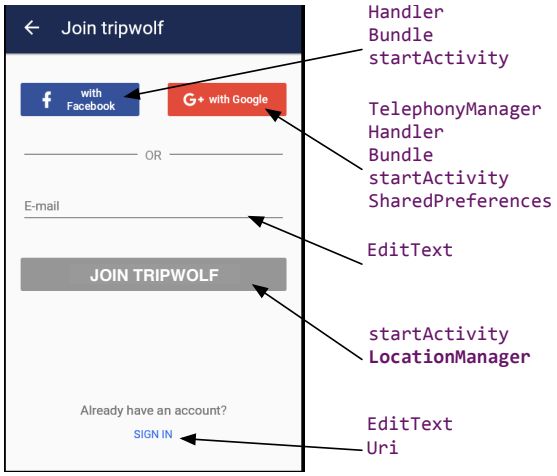

Fig. 1. For each UI element, BACKSTAGE determines the APIs it triggers, and checks for anomalies: The TRIPWOLF Signup button sends the current location to TRIPWOLF servers.

Context and APIs. For each UI element, we extract its *associated text*—both *labels* shown on the element itself, *context* from the surrounding screen, as well as the APIs that would be triggered by the UI element. In our example, "Join TRIPWOLF" uses *LocationManager.getLastKnownLocation()* to retrieve the precise current location, and *startActivity* to switch to the next input screen.

Cluster Analysis. From associated text of all UI elements, we clustered their verbs and nouns into 250 *concepts*—clusters of words with a minimal *semantical distance* using a WORD2VEC model [13]. For each UI element, we determine the distance between its text and the concepts. A button named "Share", for instance, would be semantically close to the concepts of "friend" (to share) and "finances" (a share). Figure 3 shows the label of all UI elements that form the "Signup" concept.

For each button, we also extract the APIs used. Figure 4 shows the ANDROID packages used by Signup buttons. The "normal" behavior of a Signup function is to access the network via *android.net* or *org.apache.http*. Several signup functions also access *android.telephony* to access the country code of either the current network or the inserted SIM card. The *android.location* package also is frequently used—but only to access the current local time.

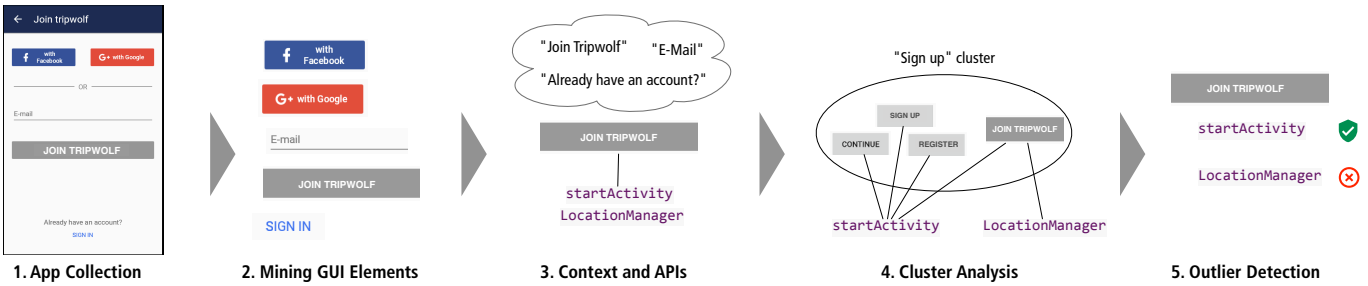

Fig. 2. How BACKSTAGE works. For each ANDROID app in a collection (Step 1), BACKSTAGE statically analyzes app code and GUI descriptions to extract UI elements (Step 2). For each UI element, it identifies its textual description and context on the screen, as well as the APIs that it triggers (Step 3). Across all apps, BACKSTAGE then clusters UI elements with semantically related descriptions APIs (Step 4) and detects outliers (Step 5)—such as the Signup button from Figure 1 that sends out the current location.

Fig. 3. Labels of semantically related UI elements, forming a "Login" concept. Based on its label and its context on the screen, the "Join TRIPWOLF" Signup button is associated with this concept.

Fig. 4. The ANDROID API packages used by the functions triggered by the UI elements in Figure 3. "Normal" functions of login buttons include accessing the Internet and switching to a new screen.

Outlier Detection. For each concept, we use outlier detection to identify those UI elements that invoke uncommon APIs, indicating differing (and possibly unexpected) behavior. Accessing the current precise location is rarely used in the Signup cluster—and thus, the TRIPWOLF Signup button is flagged as an anomaly.

The BACKSTAGE approach is in no way restricted to security and privacy issues; instead, it generalizes to *arbitrary mismatches between what a UI element shows and what it actually does.* The BACKSTAGE approach thus also can catch *simple GUI programming mistakes:* If, for instance, a programmer included a button named "Signup" that always stays on the same screen, or sends a text message, or prints a file, this error could also be caught by BACKSTAGE. Even translation issues stemming from automatic app conversion mistakes can be detected.

BACKSTAGE extends the state of the art with two important contributions:

1) In contrast to other work leveraging app collections to detect general mismatches between advertised and implemented behavior, BACKSTAGE detects anomalies at the GUI level rather than the app level. CHABADA [7] or similar approaches [17], [18], for instance, would characterize TRIPWOLF as perfectly normal, because as a travel app, it would be expected to access the current location. The fact that the location is already accessed while signing up is only detected by BACKSTAGE.
2) In contrast to existing work checking for mismatches at the GUI level, BACKSTAGE is not restricted towards stealthy behavior or privacy issues; instead, it provides a *general means to detect mismatches between advertised and implemented behavior* at the GUI level. The ASDROID approach [11], for instance, would not catch the TRIPWOLF issue, as it is neither set up for sign up processes nor location leaks nor general GUI programming mistakes.

Further contributions of the present paper include our discovery of UI elements and associated text, taking into account GUI resource files as well as the dynamic reassignment of properties of UI elements (Section II); text extraction for UI elements including context on the same screen (Section III); deep analysis of callbacks, extracting APIs and intent types (Section IV; descriptive statistics of UI element and API usage (Section V); and finally outlier detection (Section VI) and its evaluation (Section VII) introducing UI mutation analysis. The relationship to existing work is detailed in Section VIII. Section IX closes with conclusion and future work.

II. Mining UI Elements

To detect mismatches between the advertised functionality of GUI elements against their implemented functionality, BACKSTAGE has to analyze applications following three main steps:

1) identify UI elements and extract their text labels, 2) collect the *relevant* text around each UI element to better define its semantic given the context, and 3) identify the behavior that each UI element would trigger.

As easy as it may initially seem, implementing a sound technique to analyze the Android UI is not trivial, given the complexity of the Android GUI [20].

We now provide some background knowledge on the Android GUI, i.e. how to declare elements and how they can be arranged in the layout. We then proceed with explaining how we extract the information that BACKSTAGE needs.

A. Android Activities and their Layout

In the Android framework, an activity is a single screen containing several UI elements, such as buttons and text fields, organized in a hierarchy. Each app can, and typically does, contain multiple activities. The layout of the activity is usually declared in an XML file named `layout.xml`. Different files and names can be used though, as developers can bind an activity to the layout XML file thanks to the `android.app.Activity:setContentView(layoutFileId)` method. Refer to Listing 1 for an example of a layout file.

```
1  <?xml version="1.0" encoding="utf−8"?>
2    <LinearLayout android:layout_width="fill_parent"
3     android:layout_height="fill_parent">
4     <fragment android:id="@+id/fragment"
5       class="uinomaly.fragmentclass".../>
6     <Button android:id="@+id/buttonOK"
7       android:text="@string/buttonOK"
8       android:onClick="xmlDefinedOnClick"
9       style="@style/okButtonStyle"/>
10    <ImageButton android:id="@+id/imageButtonPrint" ...
11      android:src="@drawable/print_button"
12      android:contentDescription="@string/printText" />
13 </LinearLayout>
```

Listing 1. A Sample Activity layout declared in a XML file.

Besides using XML files, developers can create the entire activity layout dynamically in the app code. This strategy is rarely used, since it is error-prone and makes the maintenance of the GUI harder. Given their little prevalence, entire dynamically generated screen layouts are out of the scope of BACKSTAGE for now.

B. Dealing with Layout Reuse and Complex Elements

A layout can be entirely or partially reused in different activities in multiple ways:

`<include>` and `<merge>` XML tags. Developers can include other XML files by means of the `<include>` tag. To do this they simply have to specify the file `Id` such as `<include layout="@layout/reusableLayout"/>`. Developers can also use the `<merge>` tag to achieve the same purpose, with the advantage of eliminating redundant hierarchical elements.

Inflate layouts programmatically. `LayoutInflater` instantiates the corresponding layout file into a `View` object. The `View` object can later be added to the layout of the activity.

Fragments. Fragments are, in essence, modular sections of an activity. There are two different ways to include a fragment into an activity:

1) Declare it in the layout file of the activity directly (see lines 5–11 in Listing 1 for an example).
2) Dynamically create the fragment in the activity code by means of the `FragmentManager` class.

BACKSTAGE handles all these forms of layout reuse.

Additional challenges in the analysis of the GUI comes from dealing with complex elements such as menus, and the ANDROID framework provides several of them:

Option Menus. Option menus are placed at the top right corner on a screen. They usually give access to functionalities that are relevant for the application regardless of the context (e.g. the *Settings* button). Option menu items can be easily created by implementing the `onCreateOptionsMenu` method of the activity.

Contextual Menus. Contextual menus appear when the user presses a UI element with a long-click. They can display further actions for a specific element especially inside a `ListView`. They are created by implementing the `onCreateContextMenu` method, and they can be bound to UI elements by means of `registerForContextMenu(elementId)`.

Navigation-Drop-Down Menus. Navigation-drop-down menus are used for a quick and easy navigation through the whole application and can be identified through a small triangle in the lower right corner of the showed text. The menu items are specified through an implementation of an adapter with the according array of strings. To create such menus, the developer can invoke the `setListNavigationCallbacks(adapter,navListener)` method on an `ActionBar` instance. Even if this menu was deprecated in the Android API level 21, BACKSTAGE can still detect it, since it aims to also support apps written for old ANDROID versions.

Drawer Layouts. Drawers are panels that can be opened with a swipe from the outer vertical side of the screen to the middle. A drawer can be created with a `DrawerLayout` tag in the XML layout file.

Tab Views. Tab views are created dynamically via `actionBar.newTab()`, and can later be added to an action bar. Each tab view is represented by a fragment.

III. Extracting Text Labels from UI elements

In the previous section, we gave an overview of how activities work and how they can declare different UI elements. The goal of BACKSTAGE is to extract, for each UI element in the app, the text on its label.

There are several ways in Android to define the text of UI elements:

Label assignment in layout files. Developers can define the label of UI elements in the XML layout file by using the

TABLE I
A SET OF ATTRIBUTES RESPONSIBLE FOR BINDING TEXT TO UI ELEMENTS

android:text	android:title	android:textOn
android:hint	android:contentDescription	android:textOff
android:label		

TABLE II
A SET OF ATTRIBUTES RESPONSIBLE FOR BINDING ICONS TO UI ELEMENTS

android:background	android:drawableRight	android:drawableTop
android:src	android:drawableLeft	android:drawableBottom
android:drawableEnd	android:drawableStart	

`android:text` attribute. Refer to line 13 in Listing 1 for an example. The text can be defined either by using the reference to the app's resources with the `"@string/"` prefix or directly by providing the string that will be displayed. Even if the second option is deprecated, since it introduces localization problems, BACKSTAGE supports it. There are more attributes that can be used to set a label for a UI element (refer to Table I), and BACKSTAGE supports all of them.

Label assignment in Java code. As discussed in Section II-B, layout templates can be reused across different activities. However, the text of the UI elements in such layouts usually differs depending on the context (i.e. activity). Therefore, developers usually assign a textual label to such UI elements in the code depending on the activity. `View:setText(resourceId)` and `View:setText(text)` allow to redefine labels for UI elements. Refer to Listing 2, lines 4 and 6 for an example.

Label assignment in style files. Developers can assign labels to UI elements using the `styles.xml` file. This option is typically used when the text of UI labels changes depending on the style. Developers can specify labels of UI elements by creating an `<item>` with the attribute `name="android:text"`. Refer to Listing 1, line 15 as an example.

Beside all the aforementioned cases, which BACKSTAGE fully supports, our prototype deals with string concatenation by analyzing `StringBuilder` instances. Moreover, it performs a backward analysis from the `setText` method parameter if the relevant string is not specified there directly, but rather stored in some variables in other parts of the code.

A. Dealing with Icons

Icons are prevalent in GUIs, as they can represent the semantic of UI elements in an intuitive way. A camera icon, for instance, can be easily interpreted by a user as a button to take pictures. Icons are extensively used in mobile GUIs also because they are more space efficient than text.

A sample use of icons for UI elements is reported in Listing 1, where the `print_button` icon is bound to the `ImageButton` element. The full list of attributes responsible for binding icons to UI elements is presented in Table II.

TABLE III
A SET OF STANDARD CALLBACKS IN ANDROID

afterTextChanged	onTextChanged	onKey
onEditorAction	onClick	onDrag
onHover	onLongClick	onChronometerClick
onKeyboadDismiss	onItemClick	onItemLongClick
onItemSelected	onNothingSelected	onScroll
...	(42 more)	...

Given their prevalence, BACKSTAGE analyzes icons as well; actually, UI elements with icons make 22% of our whole dataset. BACKSTAGE handles icons by extracting *relevant text to describe them.* Icons, in fact, should also come with an alternative text, which can be read out loud to the user by a speech-based accessibility service. Developers can specify such alternative text in the `android:contentDescription` attribute of the UI element (refer to Line 12 in Listing 1). This information is what BACKSTAGE extracts and uses for UI elements that use icons instead of text labels.

B. Extracting Context of Text Labels

Well-designed applications usually have semantically meaningful text labels to improve the application usability. For example, a label "Send SMS" on a button would make it clear that the user would send an SMS message by clicking that button. However, most of the times text labels are generic in their semantic, and can only be correctly interpreted given their context. This is the case, for instance, of labels such as "OK" or "Yes", which are highly prevalent. The expected behavior for clicking an "OK" button is to confirm an operation that has been mentioned earlier or is described somewhere else in the GUI. As a consequence, together with the text label for a UI element, BACKSTAGE collects all the surrounding text, which we interpret as relevant *context* to understand the semantic of the label itself. More precisely, for each UI element we collect all the text that the activity containing the element displays.

IV. MAPPING AND ANALYZING CALLBACKS

BACKSTAGE characterizes each UI element, represented by the text label and its surrounding text, with the behavior that it would trigger at runtime. As a proxy to represent the behavior, it uses the set of the Android API invocations that are reachable, and therefore can be executed. As a preliminary step, though, it needs to identify the *callbacks*, since these are the entry points of the analysis. A callback is a special function that is bound to a particular event on a UI element that triggers its execution. The most well-known example of callback is the *onClick* function, which gets executed when the user clicks on some UI element on a screen.

There are dozens of predefined UI callbacks available in Android, but developers can also implement their own custom callbacks and bind them to any UI element. In this paper, however, we deal only with the predefined set of Android callbacks, which we report in Table III.

A. Detecting Callbacks of UI Elements

Developers can declare a callback for a UI element either statically in a *layout file* or dynamically in the *app code*.

Defining callbacks in a layout file. The most straightforward way to define callbacks is to directly declare them in the layout XML file together with the corresponding UI element. Refer to Listing 1, Line 14 for an example. However, only *onClick* callbacks can be defined this way.

Defining callbacks in code. This is common practice when UI elements are reused in different parts of the application. Developers can redefine *onClick* callbacks of UI elements by using the *setOnClickListener* method. Refer to Listing 2, Line 8 for an example. The same applies for all callbacks listed in Table III.

Similarly to Section III, UI elements on reusable layouts can have different callbacks depending on the context. To track the context of the callback, we keep track of the class name that declares it. Context is very important for binding APIs to their corresponding text for UI elements in reusable layouts.

B. Analyzing Callbacks

BACKSTAGE employs a static analysis built on top of the SOOT framework to map callbacks to APIs [19]. The analysis works along the following steps:

1) BACKSTAGE identifies *callbacks* from the UI analysis phase as discussed in Section IV-B, and sets them as entry points for the call graph construction.
2) It builds the *call graph* thanks to the Rapid Type Analysis algorithm (RTA), which limits the over-approximation by identifying those classes in the program that are possibly instantiated [6].
3) For each callback it collects all the *reachable Android API invocations* in the transitive closure of its call graph.

As discussed in Section IV-A, callbacks can be assigned to UI elements directly in the code. Beside the simple case when a single callback is bound to a single button, there are cases when one callback is bound to multiple buttons. Consider, as an example, the code in Listing 2 where the same `myClick` callback is assigned to both `okButton` and `cancelButton`. This example shows that to precisely assign API invocations to the right button the analysis should be context-sensitive, i.e. it should be able to correctly bind the `okButton` to the branch at Line 16 and the `cancelButton` to the one at Line 19 respectively.

Our BACKSTAGE prototype implements a static context-sensitive analysis that correctly handles such cases for buttons, alert dialogs, and menu items.

The code included in apk files often includes libraries. As a consequence, many API invocations that BACKSTAGE would identify with its analysis belong to third party libraries. Our initial manual evaluation of the analysis results showed that many of these library invocations are infeasible in practice. This is due to the over-approximation of static analysis. To reduce this problem, we decided to limit the analysis only on the application code, thus excluding libraries from the analysis.

```
1   @Override
2   protected void onCreate(Bundle savedInstanceState) {
3    Button okButton = (Button) findViewById(R.id.ok_button);
4    okButton.setText(R.string.okButton);
5    Button cancelButton = (Button) findViewById(R.id.cancel_button);
6    cancelButton.setText(R.string.cancelButton);
7    okButton.setOnClickListener(myClick);
8    cancelButton.setOnClickListener(myClick);
9   }
10
11  View.OnClickListener myClick = new View.OnClickListener() {
12   public void onClick(View v) {
13    switch (v.getId()) {
14     case R.id.ok_button:
15      //action if button is the okButton
16      break;
17     case R.id.cancel_button:
18      //action if button is the cancelButton
19      break;
20    }
21   }
22  };
```

Listing 2. An example of assigning the same callback to multiple buttons

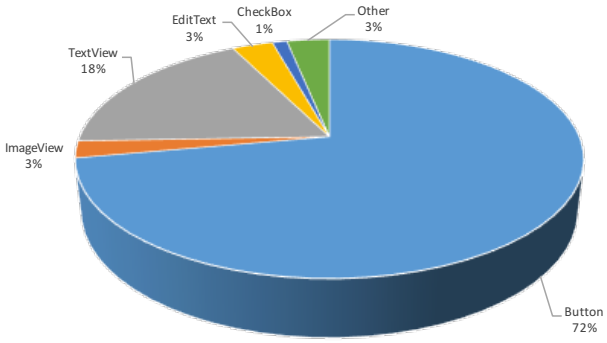

Fig. 5. Distribution of extracted UI elements

To achieve this goal, we filter classes based on their package name. Thus, for instance, when analyzing the Twitter app, we would focus only on classes belonging to the `com.twitter` package.

Furthermore, we also included a parameter to limit the depth of the call graph analysis starting from entry points. In fact, the farthest the code is from the entry points, the more likely it contains infeasible invocations. The default settings, which are what we used in our experiments, consider only invocations to the Android API that are in methods with a maximum depth of five calls from the corresponding callback.

V. The Android Apps Dataset and its UI Elements

BACKSTAGE needs a large number of apps in order to point out relevant outliers. To this end, we created a large dataset that includes the top 600 Android apps in each category of the Google Play Store as displayed in the US in July 2016. We chose the US market in order to maximize the number of apps using English as the main language. Instead of crawling the Google Play store to download the app APKs, we retrieved them from ANDROZOO [3].

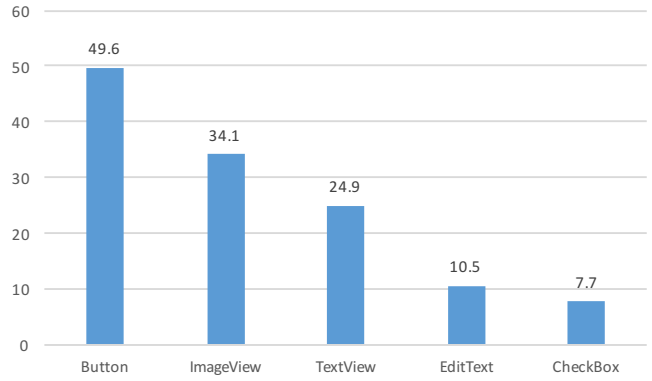

Fig. 6. Average number of reachable API invocations per UI element type

As a result of this step we collected 12,000 apps, which is less than one would originally expect. This is because either Google Play lists less than 600 top apps for some categories, or because ANDROZOO did not have the desired APK.

Given that BACKSTAGE works on detecting anomalies in the graphical user interface, we filtered out those apps with little to no GUI and those apps whose UI is mainly made of drawings on canvases (e.g. interactive games). As a heuristic to filter out the apps we could not analyze, we computed the ratio between the *number of layout files* and the *number of activities* in an app. We ignored those apps that had a ratio lower than 70%. The intuition behind this heuristic is that there is usually one layout file for each activity. When this is not the case, it means that the activity does not have a UI that can be analyzed with our approach (i.e. there are no labels with text, buttons, etc.). This heuristic, in essence led us to ignore all apps listed in the GAMES, ANDROID_WEAR, COMICS, and APP_WIDGET categories. Using this filtering heuristic allowed us to remove from our dataset those apps that are out of the scope of this work.

Moreover, given that BACKSTAGE semantically groups similar text labels, it needs to work on a dataset made of a single language. We chose English for its prevalence, and we relied on *LangDetect* [15] to detect the language of the GUI text. Similarly to the previous heuristic, we removed from our dataset those apps that had less than 70% of the resources in English. 203 apps were removed in this step.

We analyzed the remaining apps, giving a time budget of two hours for each app, and killing the process afterwards. We could not analyze 356 apps due to this timeout, and we thus removed them from the dataset too.

Despite these filtering processes, our dataset remains significantly large: We analyzed a total of 87,100 UI elements, which included 722 *unique UI element types* (Button, ImageButton, RadioButton, ToggleButton and 718 additional custom button types), and we extracted a total of 11,055 unique text labels. Figure 5 shows the distribution of UI elements in our dataset.

We analyzed more than 48,000 unique callbacks, whose analysis identified over 1 million API invocations, in which they refer to 2,602 unique APIs. Figure 6 shows the average number of method invocations that BACKSTAGE could find from each category of UI element.

Fig. 7. Wordcloud for the "share" concept

Fig. 8. Wordcloud for the "shopping" concept

VI. Detecting Outliers

Now that we do have collected all necessary information, let us now detail how BACKSTAGE detects outliers.

A. NLP Preprocessing

We first subjected all words to NLP preprocessing. From all the words in our UI elements, we retained only *verbs* and *nouns* present in the *WordNet* [14] and WORD2VEC [13] English dictionaries. We then *lemmatized* them, i.e., we grouped together the different inflected forms of a word so they can be analyzed as a single item.

B. Concepts

In any kind of anomaly detection, the first step is to determine relationships between items. For the words in our UI elements, this meant to identify those *words that are related to each other.* In our experiments, we found approaches that rely on words frequently occurring together (such as Latent Dirichlet Allocation or LDA) to be of only limited use; we attribute this to mobile screens only having a limited amount of space for text. Instead, we relied on the WORD2VEC [13] approach capturing the *semantic distance* between words. GOOGLE provides a pre-trained model based on the Google News corpus with 100 billion words ("googlenews-vectors-negative300.bin"). For each word the model produces an embedding into the high–dimensional vector space. By means of it WORD2VEC can determine, for instance, that the word "queen" is closely related to "king" and "woman".

We used WORD2VEC to determine the semantic distance between all words in UI elements, which is the cosine distance between the associated vectors. The model provided contains only a few phrases, though, labels commonly consist of more then one word. Thus, for each of the compound labels we

TABLE IV
CONCEPTS MINED FROM UI LABELS

abort · about · accept · account · account info · achievement · activate · activation · activity · add · add content · add email · add list · add photo · address · admin · agree · agreement · album · alert · alphabet · amazon · amount · answer · app · apply · appointment · apps · architecture · archive · attach · audio · authenticate · authorize · average · baby · back · background · backup · badge · bangalore · barbie · barcode · baseball · bath · beauty · bedroom · begin · birth · block · bluetooth · board · broadcast · build · business · buy · bypass · cache · calculator · calendar · call · calorie · camera · campus · cap · card · cardio · career · celsius · challenge · change · chapter · chart · check · checkout · cheer · choose · city · claim · clean · clear · click · clock · cloud · code · colombia · come · comment · commentary · connect · contact · continue · contribution · coupon · cpu · create · create account · credit · credit card · custom · customer · customize · cycle · data · day · deal · debug · decline · default · delete · demo · departure · deposit · description · desire · destination · detail · device · dictionary · do · download · draw · edit · edit account · editor · electron · email · enable · enter · error · examination · execute · export · facebook · fax · feedback · fiction · file · fill · find · folder · follower · friend · gallery · google play · handoff · health · hello · image · import · information · install · instrument · internet · invoice · itinerary · jupiter · keyboard · launch · league · license · list · location · log · login · map · meal · merge · message · mild · mode · news · next · notification · ok · open · order · panorama · password · payment · paypal · people · permission · phone · photo · picture · play · please · power usage · premium · prev · price · privacy · profile · project · projector power · pushup · quiz · redeem · register · reminder · report · reset · retry · roster · rule · save · save account · scan · scanner · search · send · setting · share · shopping · show · shutter · skip · sms · space · stay · store · submit · subscription · sync · taxi · term · test · theme · ticket · tip · title · twitter · unlock · update · upgrade · upload · url · user · vehicle · vehicle · version · view · virus · voice · wallpaper · website · weight · workout · zone

produced the aggregated vector by averaging the vectors of each word included. WORD2VEC can handle these new phrases pretty well.

We then used *soft spherical k-means clustering* [8] to cluster the labels into 250 *concepts*, listed in Table IV. As examples of such a concept, Figure 3 shows the word cloud of the "login" concept, Figure 7 shows the word cloud for the "share" concept, and Figure 8 for the "shopping" concept.

C. Membership

For each UI element, we now determine its *membership*—the proximity $p_i \in [0 \dots 1]$ of its *labels* to the concept i, where $p_i = 1$ means a perfect fit into the concept, whereas $p = 0$ implies no membership. This gives us a *membership vector* $v = (p_1, \dots, p_{250})$. We then associate each UI element e with those concepts i for which the membership is higher than a threshold of 0.5, i.e., $p_i > 0.5$. Thus, each element e is now associated with a set of concepts $C(e) = \{c_1, \dots, c_n\}$.

Some UI elements spot labels that would not be associated with any concept, resulting in $C(e) = \emptyset$. For these elements, we repeat the above process mixing the original label vectors with their *context*. More specifically, we calculate the membership for each phrase surrounding a UI element and consider ones with the maximum value. If $C(e)$ is still empty, we use the three concepts most relevant for the UI element label instead.

D. In-Concept Outliers

The next two steps combine in-category classification with overall classification, a two-stage setup first introduced with the MUDFLOW anomaly detector [5]. For each concept c_i, we determine the set of UI elements $E = \{e_1, \dots e_n\}$ that are members of c_i, i.e. $E = \bigcup_{j=1}^{250} \{e \cdot c_j \in C(e)\}$. For each UI element, we identify the set of its associated APIs. For each API, we use a string composed of class name and method name as its feature. For `ContentResolver` and `Intent` APIs, we additionally supply the associated URIs. As an example, consider Figure 4, showing the APIs used in the "Login" concept.

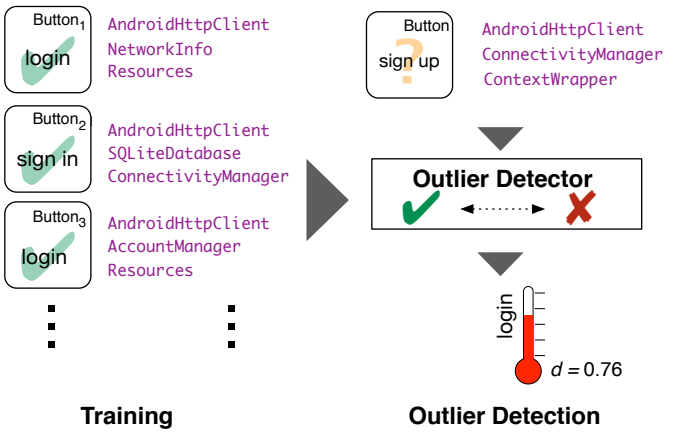

Fig. 9. Per-concept outlier detection. For each concept such as "Login", BACKSTAGE selects UI elements whose labels are related to this concept and uses their APIs as features. It then takes a new unknown UI element, and determines its outlier score with respect to the "normal" UI elements. The higher the score, the less "normal" the app behaves inside the given concept.

As illustrated in Figure 9, we then use the ORCA outlier detector to run an *outlier analysis* on the set E; by default, we consider the five nearest neighbors of a sample. To measure the dissimilarity between samples, BACKSTAGE uses the *Jaccard* distance metric, which is well-suited for data with a large number of binary features.

The resulting distance serves as an *outlier score:* The higher the distance of a sample, the less "normal" are its features. We normalize the *outlier score* as $o_i \in [0 \dots 1]$. Each UI element e is then assigned its *outlier score* o_i, representing how much e is an outlier in the concept i with respect to its APIs used.

E. Overall Classification

Across all 250 concepts, we aggregate the individual outlier scores for each UI element, assigning each element e an outlier vector ("anomaligram") $O = (o_1, \dots, o_{250})$, with o_i again being the outlier score for concept i. As illustrated in Figure 10, these vectors are then used to train a 1-class ν-SVM classifier which then can classify a vector (and thus, its associated UI element) into "likely normal" or "likely abnormal".

The result is a fully automatic warning mechanism for UI elements, which, when triggered, points developers to a mismatch between UI element labeling and the functionality it uses. Developers can resolve the mismatch either by adapting the GUI label (or context) to the functionality, or vice versa.

VII. EVALUATION

Since our analysis uses most widely used apps, with a high level of maturity, visible GUI errors as those prevented

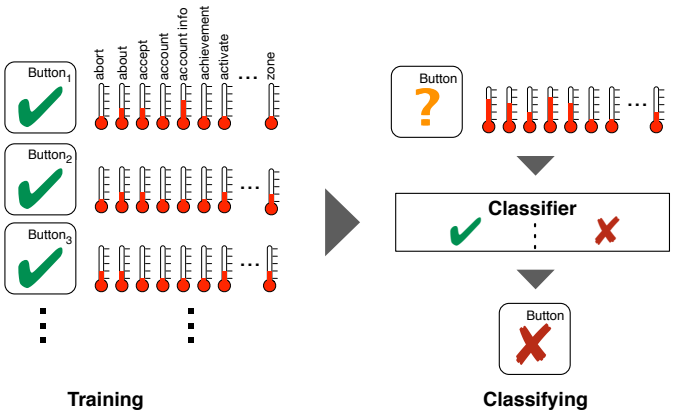

Fig. 10. Classifying UI elements across multiple concepts. For each UI element in our set, we determine its vector of probabilities of being an outlier in each concept (Figure 9). A one-class classifier trained from these vectors can label an unknown UI element as "likely normal" if it is normal across all concepts, or "likely abnormal" instead.

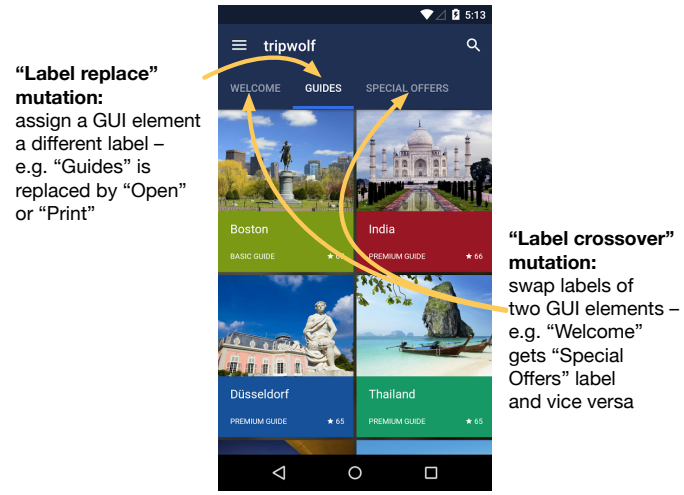

Fig. 11. GUI replace and crossover mutations as used to evaluate BACKSTAGE

by BACKSTAGE are typically quickly detected, reported, and fixed. We thus resort to a well-established scheme used to evaluate testing techniques. To evaluate how BACKSTAGE fares as it comes to *general* GUI mismatches, we devised a setting in which we would create *synthetic GUI errors* (mutations). Specifically, we would take existing buttons and change their labels such that they would no longer match the APIs used; and then evaluate whether BACKSTAGE detects these mutations as anomalies.

A. GUI Mutations

In detail, we implemented the following mutations, modeled after *mutation* and *crossover* operations in genetic algorithms:

Label replace. Given a UI element e, we replace its label with a label from another concept. This label is chosen in two ways:

- *randomly*—that is, out of all the labels encountered across all apps. This simulates a random error in labeling a UI element.
- *high distance*—that is, from a concept that is semantically distant with respect to the original label; formally, we assume a WORD2VEC semantic similarity of 0.2 or less. This avoids the problem of *equivalent mutants* known from code mutation in software testing.

Crossover. Given a UI element e, we would swap its label with the label of a random UI element $e' \neq e$ in the same app. This simulates the error of a developer confusing two UI elements, swapping callbacks to UI elements within the same application. The mismatches created by crossover would be more subtle, as they occur within the same range of app functionality.

All these mutations are applied on the data alone; we do not actually change the code of existing apps. In terms of the difference induced by the mutation, we would assume "high distance" mutations to be the easiest to detect, followed by random mutations, and finally crossover.

To the best of our knowledge, this is the first time mutation analysis is being applied to graphical user interfaces; this is a consequence of BACKSTAGE being the first approach to find general functionality mismatches in GUIs.

B. Evaluation Setting

For the evaluation of BACKSTAGE, we use the well-established procedure for *evaluating binary classifiers*. In detail:

1) We start with the set A of all mined UI elements. As these come from mature apps with millions of users, we assume the UI elements are almost all correct.
2) We create a random subset $A' \subset A$ with $|A'| = 90\% \cdot |A|$ and train BACKSTAGE from A'.
3) We create the set $C \subseteq A$ with $C \cap A' = \emptyset$ and consequently $|C| = 10\% \cdot |A|$. This will be our set of "known correct" UI elements.
4) We create a set of mutants M derived from A, such that $|M| \approx 10\% \cdot |A| = |C|$. This will be our set of "known incorrect" UI elements. Both mutation types are applied with equal likelihood.
5) We create the testing set $T = M \cup C$ and have these UI elements classified by BACKSTAGE.

This process is repeated five times. We assess the accuracy of the BACKSTAGE classifier by means of standard metrics such as *precision* (how many of the reported anomalies would be mutants?) and *recall* (how many of the mutants would be reported as anomalies?); all values would be mean values over the five repetitions.

C. Results

The overall results of BACKSTAGE are summarized in the confusion matrices in Table V, Table VI, and Table VII. Let us discuss random mutations first, and then contrast the results with the alternative mutation schemes.

As seen in Table V, of the 4,999 mutants fed into BACKSTAGE, 3,369 are correctly classified as being abnormal, which

TABLE V
BACKSTAGE ACCURACY FOR "RANDOM" LABEL REPLACE MUTATIONS.

Input	Classified as: Abnormal	Normal	Total	
Mutant	TP = 3369	FN = 1630	4999	*Precision* = 75%
Correct	FP = 1100	TN = 4056	5156	*Recall* = 67%
Total	4469	5686	10155	*Accuracy* = 73%
				Specificity = 79%

TABLE VI
ACCURACY FOR "HIGH DISTANCE" LABEL REPLACE MUTATIONS.

Input	Classified as: Abnormal	Normal	Total	
Mutant	TP = 3528	FN = 1471	4999	*Precision* = 76%
Correct	FP = 1096	TN = 4060	5156	*Recall* = 71%
Total	4624	5531	10155	*Accuracy* = 75%
				Specificity = 79%

TABLE VII
BACKSTAGE ACCURACY FOR CROSSOVER LABEL MUTATIONS.

Input	Classified as: Abnormal	Normal	Total	
Mutant	TP = 2290	FN = 2475	4765	*Precision* = 69%
Correct	FP = 1026	TN = 4121	5147	*Recall* = 48%
Total	3316	6596	9912	*Accuracy* = 65%
				Specificity = 80%

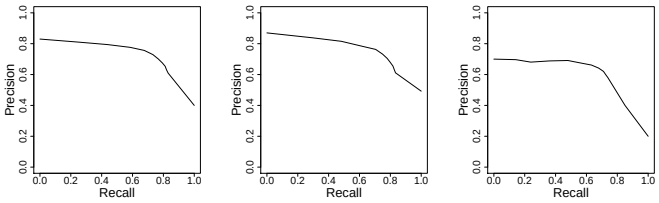

Fig. 12. Precision vs. recall curves for "random" (left), "high distance" (middle), and "crossover" (right) mutation types

results in a recall rate of $3369/4999 = 67\%$. As expected, high distance mutations (Table VI) have an even higher chance (71%) of being detected. Even if the programmer confuses two buttons (Table VII), BACKSTAGE detects every second such mistake. All these results should be interpreted from the standpoint that to the best of our knowledge, there is no other approach which would detect such mismatches.

A high recall means little, though, if the precision is low; that is, if the UI elements reported by BACKSTAGE contain many false positives. For "random" mutations (Table V), the precision is 75%, meaning that three out of four UI elements reported will actually be abnormal; high distance mutations fare even slightly better, with 76%. For "crossover" mutations (Table VII), BACKSTAGE still has a precision of 69%; a bit more than two out of three UI elements reported will be true anomalies. This high precision makes BACKSTAGE a practical tool. As shown in Figure 12, one can trade an even higher precision for lower recall and vice versa by choosing alternative anomaly thresholds.

So what are the limitations of BACKSTAGE at this point? Obviously, the semantic distance between the given and the correct label is crucial. If this distance is high, as indicated by our "high distance" mutations, BACKSTAGE can easily find them. But if this distance is low, BACKSTAGE fails. Errors that BACKSTAGE misses typically would be semantically close to each other and not differ in the APIs used—for instance, "Search with Google" and "Search with Bing" will use the same APIs, namely interacting with a remote server. If the labels are semantically almost equivalent, as in "Stop" vs "Abort" vs "Cancel", BACKSTAGE will have a hard time differentiating them, just as humans will.

D. Threats to Validity

As any empirical study, our evaluation is subject to multiple threats to validity. In terms of *external validity,* the biggest threat is our mutation model, which may or may not be representative for real programming errors, as far as GUIs are concerned. We are not aware of a comprehensive source for this kind of data, and thus must speculate.

In terms of *construct validity,* there are several parts of our analysis that induce noise. Our static analysis may over-approximate and report APIs to be part of the callback that cannot be reached in actual executions. Likewise, our anomaly detection model may overgeneralize, and thus come to false conclusions; this is what our mutation model addresses, using standard measures such as precision and recall.

VIII. RELATED WORK

Our work is related to three central strands of work, and in each, there is one paper that stands out by having very much inspired and influenced this work. In detail:

Detecting UI anomalies. ASDROID [11] by Huang et al. is the first work to explicitly analyze mismatches between user interfaces and program behavior. Its general setting is similar to BACKSTAGE, in the sense that it maps UI elements to invoked functions, and checks labels as well as APIs invoked. However, it only checks for a small set of fixed scenarios, such as sending text messages or making phone calls in the background, both in terms of labels analyzed as well as in the set of invoked APIs. This is because ASDROID focuses on stealthy (malicious) behavior only.

BACKSTAGE can be seen as a generalization of ASDROID. By mining thousands of UI elements, BACKSTAGE can detect *arbitrary mismatches* between user interfaces and associated behavior. Such mismatches include stealthy behavior as detected by ASDROID: On mature apps with well-tested user interfaces, most anomalies found by BACKSTAGE would show stealthy behavior, as this would not be found during GUI testing. However, such mismatches also include misleading button labels, wrong API associations and more; and all these would be discovered by BACKSTAGE only.

Mining apps. The CHABADA [7] work by Gorla et al. was the first work to explicitly detect arbitrary mismatches between app descriptions and app behavior. The key idea of CHABADA, mining app stores to analyze and classify thousands of apps has since fueled a new research field, namely app mining [1].

The CHABADA approach comparing descriptions and APIs is well reflected in the BACKSTAGE setting—except that CHABADA classifies at the app level, whereas BACKSTAGE classifies at the UI element level. BACKSTAGE can thus be seen as a more fine-grained specialization of CHABADA and similar approaches [17], [18], [24], [2], [23], [12]. By working at the UI element level, BACKSTAGE can detect abnormal behavior that would be missed by CHABADA. In our introductory TRIPWOLF example, where a Signup button accesses the precise user location, CHABADA would have missed the anomaly, because at the app level, it is perfectly normal for a travel app to access the precise location. For a Signup function, however, it is not; and thus, BACKSTAGE can detect the problem. The specialization on UI elements also allows BACKSTAGE to find many more mistakes associated with bad GUI programming.

Relating UI elements and effects. GATOR [21] by Yang et al. was the first work to provide precise mappings between ANDROID UI elements and their callbacks via a pure static analysis. Earlier work had focused on dynamic analysis and exploration, either in a black box [4] or a grey box [22] style. The advantage of static analysis is that it can explore and identify UI elements that would be hard to reach dynamically—because accessing them would require, say, a password, an in-app purchase, or the defeat of a boss monster.

The analysis in BACKSTAGE follows the GATOR approach in creating such mappings. However, we also address the specific need to extract the visible *text* and *context* from the UI elements, as well as to identify *dynamic changes* of text, context, and callbacks. Our analysis can thus be interpreted as a specialization of GATOR towards text extraction.

UIPicker [16], SUPOR [9] and BIDTEXT [10] also analyze UI elements of Android apps. However they do so to automatically identify sensitive user inputs and sensitive data disclosure. Their final aim, thus, is quite different from ours.

One field that is missing in the above list is *Human-Computer Interaction* (HCI). Interestingly, to the best of our knowledge (and to the knowledge of HCI experts in the field), *we are not aware of any work in HCI that would rely on large-scale mining and analysis of UI elements.* BACKSTAGE thus opens the door for general *automatic anomaly detection* in user interfaces, considering features such as their visual appearance, their natural language semantics, their layout, their interaction, or their behaviors; and we see plenty of future potential in this direction.

IX. Conclusion and Future Work

BACKSTAGE is the first work to generally check the *advertised* functionality of UI elements against their *implemented* functionality. To this end, BACKSTAGE analyzes thousands of existing UI elements for text and context shown to the user, clusters them by common concepts, and in each cluster, detects *outliers*—that is, UI elements that use different APIs than the others. This approach is general and effective: In our evaluation, BACKSTAGE was able to effectively identify GUI behavior mismatches with high accuracy.

Despite these advances, we see BACKSTAGE not as an end—but rather as a beginning of a new field "GUI mining", where we would mine thousands of GUIs to learn common vs. uncommon features in behavior, appearance, and process. Our future work will focus on the following topics:

Dynamic behavior. Despite the ease of static analysis, we are considering using additional dynamic analysis and exploration to assess dynamic features. Most notably, we want to *validate* anomalies as reported by BACKSTAGE by creating test cases that demonstrate the actual API access.

Automatic repair. Detecting that a UI element label does not match its behavior allows for automatic suggestions of better fitting labels. One idea we are investigating is to identify labels of UI elements that use similar APIs and to suggest them as automatic repairs: "This button should be named 'Send'."

Alternate domains. Besides ANDROID apps, there are several other domains with programs whose GUIs could be mined, such as desktop applications.

Alternate GUI features. Besides looking for anomalies between text and behavior, one might also examine anomalies in visual presentation ("The 'Send' button should be highlighted"), layout, process, or visual images—opening the door to general automatic anomaly detection and recommendations for GUI design.

To facilitate assessment, reproduction, and extension, all of BACKSTAGE is publicly available—from source code to build instructions to evaluation scripts. For details, check out the project site at

http://www.st.cs.uni-saarland.de/appmining/backstage/

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, and by the Madrid Regional Government project *N-Greens Software* (grant n. S2013/ICE-2731). Special thanks go to Curd Becker and Nikolas Havrikov for their technical support and helpful comments on earlier revisions of this work.

References

[1] A. Al-Subaihin, A. Finkelstein, M. Harman, Y. Jia, W. Martin, F. Sarro, and Y. Zhang. App store mining and analysis. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, pages 1–2, New York, NY, USA, 2015. ACM.

[2] 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.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of Android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[5] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden. Mining apps for abnormal usage of sensitive data. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 426–436, 2015.

[6] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[7] 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.

[8] K. Hornik, I. Feinerer, M. Kober, and C. Buchta. Spherical k-means clustering. *Journal of Statistical Software*, 50(10):1–22, 2012.

[9] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium*, USENIX-SEC'15, pages 977–992, Berkeley, CA, USA, 2015. USENIX Association.

[10] J. Huang, X. Zhang, and L. Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '16, 2016. to appear.

[11] 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.

[12] K. Kuznetsov, V. Avdiienko, A. Gorla, and A. Zeller. Checking app user interfaces against app descriptions. In *Proceedings of the 1st International Workshop on App Market Analysis (WAMA)*, WAMA '16, 2016.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[14] G. A. Miller. Wordnet: A lexical database for English. *Commun. ACM*, 38(11):39–41, Nov. 1995.

[15] S. Nakatani. Language detection library for java, 2010.

[16] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *Proceedings of the 24th USENIX Conference on Security Symposium*, USENIX-SEC'15, pages 993–1008, Berkeley, CA, USA, 2015. USENIX Association.

[17] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.

[18] 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 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1354–1365, New York, NY, USA, 2014. ACM.

[19] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[20] Y. Wang, H. Zhang, and A. Rountev. On the unsoundness of static analysis for Android GUIs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 18–23, New York, NY, USA, 2016. ACM.

[21] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 89–99, Piscataway, NJ, USA, 2015. IEEE Press.

[22] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265, Berlin, Heidelberg, 2013. Springer-Verlag.

[23] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 303–313, Piscataway, NJ, USA, 2015. IEEE Press.

[24] 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.