Analyzing the User Interface of Android Apps

Konstantin Kuznetsov[♣] · Vitalii Avdiienko[♣] · Alessandra Gorla[♠] · Andreas Zeller[♣]

[♣]CISPA, Saarland University, Saarbrücken, Germany

[♠]IMDEA Software Institute, Madrid, Spain

ABSTRACT

When interacting with Android apps, users may not always get what they expect. For instance, when clicking on a button labeled "*upload picture*", the app may actually leak the user's location while uploading photos to a cloud service. In this paper we present BACKSTAGE, a static analysis framework that binds UI elements to their corresponding callbacks, and further extracts *actions* in the form of Android sensitive API calls that may be triggered by events on such UI elements. We illustrate the inner workings of the analysis implemented by BACKSTAGE and then compare it against similar frameworks.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**;
• **Theory of computation** → *Program analysis*;

ACM Reference Format:
Konstantin Kuznetsov[♣] · Vitalii Avdiienko[♣] · Alessandra Gorla[♠] · Andreas Zeller[♣]. 2018. Analyzing the User Interface of Android Apps. In *Proceedings of MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , Gothenburg, Sweden, May 27–28, 2018 (MOBILESoft '18),* 4 pages.
https://doi.org/10.1145/3197231.3197232

1 INTRODUCTION

Users interact with Android mobile apps through user interfaces, but it is often unclear whether the actual underlying behavior of an app reflects the users' expectations. There may be many reasons why the actual behavior differs from the expected one: App developers may intentionally hide some undesired behavior from the user to secretly collect sensitive information. The actual behavior may differ from the expected one even when app developers have good intentions but have little experience with good UI design. As an example, consider Figure 1, showing a menu in the Android *Health Tracker Lite* app. Users can export their data to a CSV file, resulting in a "file saved" message confirming the successful export. However, when users click the "Send User Data By Email" button, they obtain *the very same message.* Just a few seconds later, a mail dialog pops up that allows to send the saved file, but the message

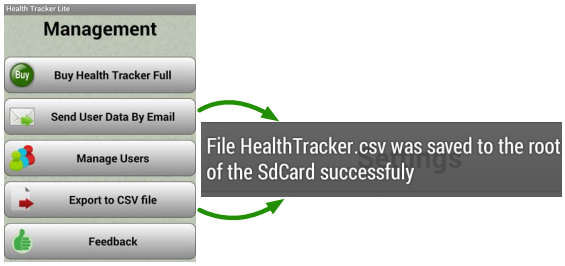

Figure 1: Health Tracker Lite app. The same message "File saved" is shown for export as well as for the email actions.

is still confusing. Finally, the actual behavior may differ from what users expect because of an implementation bug in the app. Whatever the scenario and whatever the motivation might be, in order to fully test, analyze, or simply understand the behavior of Android apps there needs to be a technique that can analyze the app's UI and the code associated to it as a whole.

The challenges include:

Obtaining the set of UI elements. In Android apps, many UI elements are statically declared in a layout file but can also be created, enabled, or disabled dynamically in code.

Modeling UI control flow. The control flow of Android apps is determined by the lifecycle and interactions of the UI elements, which are defined by means of associated event handling callbacks.

Obtaining UI element contents. Contents of UI elements may also be defined or updated from code, again calling for analysis of callback code.

This data could be obtained dynamically, using *UI event generation* to systematically explore the user interface of an app—but then one would struggle to achieve high coverage [3]. In this paper, we present BACKSTAGE, a static analysis framework that runs the following analyses on a given Android app:

Analysis of UI Elements. BACKSTAGE determines all UI elements that are declared in the app, either in the layout file, or created dynamically from within lifecycle methods. For the *Health Tracker* app, for instance, BACKSTAGE determines the UI buttons shown in Figure 1, as they would be declared in the layout file.

Analysis of UI control flow. By associating and analyzing the *callback functions* that would be activated by a UI element, BACKSTAGE can determine how UI elements invoke or activate each other. In the *Health Tracker* app, for instance, BACKSTAGE finds that the "Send User Data by Email" and "Export to CSV file" buttons both invoke a notification message.

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05.
https://doi.org/10.1145/3197231.3197232

Analysis of reachable Android API calls. Using the identified callbacks as entry points, BACKSTAGE can determine which *sensitive* Android API calls can be reached. We refer to sensitive API calls as the subset of the whole Android framework API that can perform concerning operations, such as accessing users' sensitive data (e.g. precise location and phone number), sending text messages, or making phone calls.

With these analyses in place, BACKSTAGE can 1) identify buttons such as "Send User Data by Email" and "Export to CSV file", 2) retrieve their corresponding labels (even when dynamically set), 3) identify that the callbacks associated to the buttons both lead to the temporary "File saved" message visible in Figure 1, and a `FileWrite` API call. When such message appears, the app executes a write operation to the device SD card, which is one of the sensitive API calls that BACKSTAGE considers.

The association of natural language text of UI elements to API calls opens several possibilities for mining and analyzing apps. BACKSTAGE, for instance, can be used to detect stealthy behavior [1] (similar to [4]), or to detect usability issues, as in the case of the Health Tracker app.

BACKSTAGE can provide a complete association between both UI elements, their code, and their contents, whether declared statically or created and updated dynamically.

2 ANALYZING UI AND CODE

BACKSTAGE takes as input the APK of an Android app, which includes the bytecode and resources such as the *layout files* that declare the individual UI elements in each *activity*. It produces as *output* the set of UI elements, each element e associated with:

- the parent of e in the UI hierarchy tree, as well as the activity, to which element e belongs;
- the visible label of e in natural language text or the icon;
- the callbacks associated with e, including the APIs that may be reached from such callbacks;
- other UI elements, e.g. notification messages, that would be activated as a result of activating e.

To implement this analysis, BACKSTAGE follows three main steps: 1) It retrieves the set of UI elements that the app defines and identifies the callbacks that are associated to them (Section 2.1), 2) Using the callbacks as entry points, it retrieves the list of reachable sensitive Android API calls (Section 2.2), and 3) It analyzes the content of UI elements to collect the natural language text associated with them (Section 2.3). None of these analyses is trivial, given the complexity of the Android GUI [7].

2.1 Mapping UI Elements to Callbacks

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 may contain multiple activities (and typically does so). The layout of the activity is usually declared in XML files residing in `layout` folder. Developers can bind an activity to a layout XML file using the *Activity::setContentView(layoutFileId)* method. As an example of a layout file, consider Listing 1.

BACKSTAGE parses the XML layout files to determine the set of declared UI elements, together with their callbacks and their content. BACKSTAGE supports all common techniques of reusing layouts

```
1 <?xml version="1.0" encoding="utf−8"?>
2 <LinearLayout style="@style/layout_background"...>
3   <Button android:id="@id/send_button" android:text="@string/
      ↪ send_button" android:drawableLeft="@drawable/icon_send"/>
4   <Button android:id="@id/about" android:text="About"
      ↪ android:onClick="showAbout"/>
5 </LinearLayout>
```

Listing 1: Extract from XML layout file of Health Tracker Lite

in different contexts, together with the inclusion and merging of layout files, as well as fragments. BACKSTAGE also handles complex UI elements combined in menus (such as multiple options, drop-downs, and context menus), drawer layouts and tab views.

In Listing 1 we can see how the association between UI elements and code takes place. Each UI element has an *identifier* (such as *@id/send_button*) that allows the code to refer to it, to activate it, or to update its content. UI elements also may have *text*, which typically comes as a symbolic label (such as *@string/send_button*) that would be replaced by a string according to the user's language. Finally, UI elements are tied to *callbacks*—functions that would be invoked when the UI element is activated. For the *@id/about* button, the *onClick* callback is defined statically in the XML. When the button is clicked, the method *public void showAbout(View v)* is invoked, with v being the button itself.

Developers can dynamically bind an *onClick* callback to a UI element by using the *setOnClickListener(View.OnClickListener)* method. A similar procedure applies for all the 57 callbacks that the Android framework offers [1]. BACKSTAGE can correctly associate all of these callbacks with UI elements when they are assigned dynamically, as in Listing 2.

Context Sensitivity. The analysis implemented by BACKSTAGE is context- and object-sensitive. It can precisely propagate UI objects and bind callbacks, including the ones overridden in subclasses or defined by means of anonymous inner classes. In Android, multiple UI elements may share the same callback, when the program flow is controlled by switch constructs. To correctly associate callbacks to UI elements, BACKSTAGE examines each method that is reachable from a particular callback and analyzes the branch based on the *identifier* of the UI element. Currently it supports switch cases but does not handle arrays and assignments inside a loop.

Dealing with Fragments. Fragments are modular sections of an activity. A fragment has its own lifecycle and responds to its own input events. According to the official Android documentation, there are two different ways to include a fragment into an activity:

```
1 public class com.benoved.phr_lite.manage_activity{
2   public void showAbout(View view){...}
3   @Override
4   public void onCreate(android.os.Bundle bundle){
5     Button sendUserData = (Button)findViewById(2131624047);
6     sendUserData.setOnClickListener(new onClickListener(){
7       public void onClick(View v) {...}
8     });
9   }}
```

Listing 2: Callback assignment to the "Send User Data By Email" button.

[1] https://developer.android.com/guide/topics/ui/ui-events.html

statically, within a layout file, and dynamically by means of the `FragmentManager` class. A notable feature of using fragments in an activity is the ability to add, remove, replace, and perform other actions with them in response to user actions. Each set of changes committed to the activity is called a transaction, and it can be performed by means of APIs in `FragmentTransaction` class. Listing 3 shows an example on how to replace the `my_button` element with the content of the `MyFragmentClass` at runtime.

```
1 FragmentManager fragmentManager = getFragmentManager();
2 FragmentTransaction fragmTransaction = fragmentManager.beginTransaction();
3 Fragment fragment = new MyFragmentClass();
4 fragmTransaction.add(R.id.myButton, fragment);
5 fragmTransaction.commit();
```

Listing 3: Manipulating fragments at runtime.

Dealing with fragments is necessary to properly analyze the UI of Android apps. BACKSTAGE treats fragments as special activities, and performs the same analysis. Thus, in presence of statically declared fragments, i.e., when a fragment is declared inside a layout file, BACKSTAGE treats them as `include` tags. Dynamically added fragments, instead, require to statically analyze the code. BACKSTAGE searches for all method signatures of the `FragmentTransaction` class that append a fragment to an activity: `add()` and `replace()`. It then performs an inter-procedural reachability analysis to identify the *id* of the inflated fragment container.

2.2 Analyzing User Interface Code

What happens when events on UI elements trigger callbacks? And how do these callbacks in turn affect the UI elements? To this end, BACKSTAGE employs a static analysis built on top of the SOOT framework to map callbacks to sensitive Android APIs [6]. The analysis works along the following steps:

(1) BACKSTAGE identifies the *callbacks* from the UI analysis phase (as discussed in Section 2.1) and sets them as entry points for the call graph construction.
(2) It builds the *call graph* using the Rapid Type Analysis algorithm (RTA), which limits the over-approximation by identifying those classes in the program that are possibly instantiated [2].
(3) For each method reachable from a callback, BACKSTAGE collects the list of sensitive Android API [5] invocations including the ones that activate user notifications: toast messages, alert dialogs, or push notifications.

Many API invocations that BACKSTAGE identifies may belong to third party libraries, which are often included in Android apps. Some analyses may not be interested in library code, and as a consequence BACKSTAGE provides a parameter to *limit the analysis only to the application code,* thus excluding libraries and speeding up the process. To achieve this goal, we filter classes based on their package name prefix. Thus, for instance, when analyzing the Twitter app, we would focus only on classes belonging to the `com.twitter` package. To achieve a higher scalability, BACKSTAGE also includes a parameter to limit the depth of the call graph analysis. The farther the code is from the entry points, the more likely it contains infeasible invocations. The default settings consider only invocations to the Android API that are in methods with a maximum depth of five calls from any callback [2].

[2]based on the median value of 1000 random apks analyzed

2.3 Analyzing User Interface Content

The final step in the BACKSTAGE analysis is to determine the text associated with each UI element regardless of whether it is statically or dynamically generated. In Android, there are several ways to define the text of UI elements, and BACKSTAGE supports all of them:

Content assignment in layout files. Developers can define the content of UI elements in the XML layout file by setting the `android:text` attribute. 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. BACKSTAGE supports both options. In our *Health Tracker* example, BACKSTAGE determines the content of a button such as "Send User Data By Email" by extracting the string from the *android:text* attribute of the corresponding button.

Content assignment in style files. In order to specify the appearance for a single UI element developers can create a style, which is a collection of UI attributes. All styles should be declared in the `styles.xml` resource file. Beside various attributes such as font size and font color, a style can also define the text of UI labels by providing the `android:text` attribute.

Content assignment from code. Static layout templates can be reused across different activities. However, the text of the UI elements in such layouts usually differs depending on the context. Therefore, developers usually assign a textual label to such UI elements in the code depending on the usage. Android methods `View:setText(resourceId)` and `View:setText(text)` allow to redefine the content for UI elements. Since text content is frequently dynamically computed from other strings and values, BACKSTAGE performs an *intra-procedural backward analysis* from the `setText` method parameter if needed. BACKSTAGE also analyzes string concatenations from `StringBuilder` instances to identify the desired text. One UI element can be reused several times, for instance, in fragments. BACKSTAGE tracks these re-definitions and reports all values collected, each one supplied with the name of the class where it was changed. Therefore, the UI can be correctly mapped to the corresponding callback.

Content from icons and graphics. Many UI elements use icons rather than text. This way, they can represent the semantic of UI elements in an intuitive and space efficient way. BACKSTAGE handles icons by extracting both icon names (such as `icon_send`) and *alternative text*, which is often provided for speech-based accessibility services. Developers can specify such alternative text in the `android:contentDescription` attribute of the UI element.

With these analyses BACKSTAGE knows the UI elements declared in an app, the content of all UI elements, such as the buttons and messages, and which buttons invoke which APIs, which in turn invoke which messages with which content.

3 EVALUATION

As a preliminary evaluation of BACKSTAGE we aimed to compare its abilities against the latest version of GATOR (November 2017) [8]. In essence, the goals of BACKSTAGE and GATOR are similar since they both aim to create a precise mapping between UI elements and their corresponding callbacks. However, BACKSTAGE also addresses the specific need to extract natural language text from UI elements,

Table 1: User Interface Elements Support

UI Element	GATOR	BACKSTAGE
<include>xml tag	✓	✓
LayoutInflater	✓	✓
<fragment>xml tag	✗	✓
FragmentManager	✗	✓
Option menu	✓	✓
Pop-up menu	✗	✓
Contextual menu	✓	✓
Navigation-Drop-Down menu	✗	✓
Drawer Layout	✓	✓
Tab View (via ActionBar)	✗	✓
ViewPager		
xml defined	✓	✓
via PagerAdaptor	✗	✓
AlertDialog	✓	✓
Toast	✗	✓
Notification	✗	✓

Table 2: Listeners Support

Listener Type	GATOR	BACKSTAGE
Layout xml file	✓	✓
Style xml file	✗	✓
Anonymous inner class	✶	✓
Menu callbacks	✓	✓
Listener interface	✶	✓
Shared superclass	✶	✓
Listener assigned in a loop	✶	✗

✓— listener correctly identified, ✗— listener not found,
✶— too many listener found, over-approximation

as well as to identify which sensitive API calls can be reached from such UI elements. We first ran BACKSTAGE and GATOR on a small set of synthetic examples we crafted to cover many different features of the Android UI. We present the results of our analysis in Section 3.1. We conclude our comparison by briefly presenting the results on the Health Tracker app in Section 3.2.

3.1 Comparison on Synthetic Samples

To compare which features are supported by BACKSTAGE and GATOR, we carefully reviewed the Android documentation and crafted a set of synthetic samples to validate the capabilities of each tool. Table 1 and Table 2 report the results of our analysis. It is worth to note that GATOR does not support *fragments*, which are now widely used by many developers. Thus, all UI elements residing in fragments remain unexplored. GATOR correctly identified the half of the menus available in Android. Finally, it has limited support of dialogs and notifications. Moreover, GATOR tends to over-approximate in most cases (see Table 2), associating too many listeners to UI elements.

3.2 Comparison on the Health Tracker app

On the HealthTracker app, GATOR reported *9 listeners* bound to the *Send User Data By Email* button. It assigned the exact same set of listeners to each of the 7 buttons of the `Management` screen. Two additional listeners come from buttons of the *Dialog* invoked by one of the buttons. This dialog was correctly identified by GATOR, but the list of listeners is over-approximated. Thus, for the Health Tracker app GATOR could produce only a rough model of the activity, missing many details. In contrast, BACKSTAGE generated a precise model of the same activity. It correctly assigned all listeners to the corresponding buttons. Along with handler mapping, it supplied each button with its label, and produced a list of APIs reachable from each listener.

The major cause of over-approximation in UI hierarchy construction and callback handler mapping of GATOR is the missing support of dynamic dispatch recognition. For instance, for a set of subclasses, it reports a union of all layouts assigned by means of a method defined in a shared superclass, whereas the *layout id* is redefined inside each subclass. The same over-approximation also happens for listeners implemented by means of anonymous inner classes. In particular, GATOR can not distinguish listeners declared via anonymous inner classes, i.e. if two UI elements have different listeners it will assign both to each of them.

4 CONCLUSION

In this paper we presented BACKSTAGE, a static analysis framework to analyze the UI of Android apps. We listed challenges pertaining to this domain, and we compared our tool with the state of the art showing that BACKSTAGE can solve some of the current shortcomings that other techniques have.

The code of BACKSTAGE is open source and available at:

https://github.com/uds-se/backstage

ACKNOWLEDGMENTS

This work was supported by the EU FP7-PEOPLE-COFUND project AMAROUT II (n. 291803), by the European Research Council, project SPECMATE, by the Spanish project DEDETIS, and by the Madrid Regional project *N-Greens Software* (n. S2013/ICE-2731).

REFERENCES

[1] V. Avdiienko, K. Kuznetsov, I. Rommelfanger, A. Rau, A. Gorla, and A. Zeller. Detecting behavior anomalies in graphical user interfaces. In *ICSE 2017: Proc. of the 39th Intl. Conf. on Software Engineering Companion*, pages 201–203, 2017.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, 1996.

[3] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *ASE 2015: Proc. of the 30th Annual Intl. Conf. on Automated Software Engineering*, pages 429–440. IEEE Computer Society, 2015.

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

[5] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *NDSS 2014: 20th Annual Symposium on Network and Distributed System Security*, 2014.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In *CASCON*, pages 13–23. IBM Press, 1999.

[7] Y. Wang, H. Zhang, and A. Rountev. On the unsoundness of static analysis for Android GUIs. In *SOAP 2016: Proc. of the 5th ACM SIGPLAN Intl. Workshop on the State Of the Art in Java Program Analysis*, pages 18–23, 2016.

[8] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE 2015: Proc. of the 37th Intl. Conf. on Software Engineering*, pages 89–99. IEEE Press, 2015.