2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)

StoryDroid: Automated Generation of Storyboard for Android Apps

Sen Chen*, Lingling Fan*, Chunyang Chen†, Ting Su‡, Wenhe Li§, Yang Liu‡, Lihua Xu§
*East China Normal University, China †Monash University, Australia
‡Nanyang Technological University, Singapore §New York University Shanghai, China
ecnuchensen@gmail.com

***Abstract*—Mobile apps are now ubiquitous. Before developing a new app, the development team usually endeavors painstaking efforts to review many existing apps with similar purposes. The review process is crucial in the sense that it reduces market risks and provides inspiration for app development. However, manual exploration of hundreds of existing apps by different roles (e.g., product manager, UI/UX designer, developer) in a development team can be ineffective. For example, it is difficult to completely explore all the functionalities of the app in a short period of time. Inspired by the conception of storyboard in movie production, we propose a system, StoryDroid, to automatically generate the storyboard for Android apps, and assist different roles to review apps efficiently. Specifically, StoryDroid extracts the activity transition graph and leverages static analysis techniques to render UI pages to visualize the storyboard with the rendered pages. The mapping relations between UI pages and the corresponding implementation code (e.g., layout code, activity code, and method hierarchy) are also provided to users. Our comprehensive experiments unveil that StoryDroid is effective and indeed useful to assist app development. The outputs of StoryDroid enable several potential applications, such as the recommendation of UI design and layout code.**

***Index Terms*—Android app, Storyboard, Competitive analysis, App review**

I. Introduction

Mobile apps now have become the most popular way of accessing the Internet as well as performing daily tasks, e.g., reading, shopping, banking and chatting [23]. Different from traditional desktop applications, mobile apps are typically developed under the time-to-market pressure and facing fierce competitions — over 3.8 million Android apps and 2 million iPhone apps are striving to gain users on Google Play and Apple App Store, the two primary mobile app markets [24].

Therefore, for app developers and companies, it is crucial to perform extensive competitive analysis over existing apps with similar purposes [11, 34, 54, 55]. This analysis helps understand the competitors' strengths and weaknesses, and reduces market risks before development. Specifically, it identifies common app features, design choices, and potential customers. Moreover, researching similar apps also helps developers gain more insight on the actual implementation, given that delivering commercial apps can be time-consuming and expensive [18].

Typically, to achieve the aforementioned analysis, a freelance developer or a product manager (PM) in a tech company has to download the apps from markets, install them on mobile devices, and use them back-and-forth to identify what she is interested in [11, 34, 54, 55]. However, such manual exploration can be painstaking and ineffective. For example, if a tech company plans to develop a social media app, 245 similar apps on Google Play will be under review. It is overwhelming to manually analyze them — register accounts, feed specific inputs if required, and record necessary information (e.g., what are the main features, how are the app pages connected). Additionally, commercial apps can be too complex to manually uncover all functionalities in a reasonable time [36]. For UI/UX designers, the same exploration problem still remains when they want to get inspiration from similar apps' design. In addition, the large number of user interface (UI) screens within the app also makes it difficult for designers to understand the relation and navigation between pages. For developers who want to get inspiration from similar apps, it is difficult to link the UI screens with the corresponding implementation code — the code can be separated in static layout files as well as a large piece of functional code.

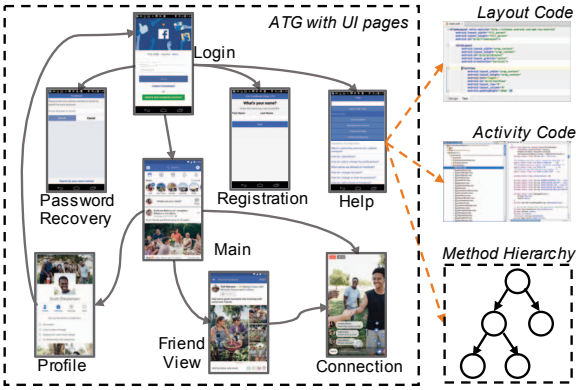


Fig. 1: The storyboard diagram of an Android app

Inspired by the conception of *storyboard*[1] in movie industry [53], we intend to generate the *storyboard* of an app to visualize its key app behaviors. Specifically, we use activities (i.e., UI screens) to characterize the "scenes" in the storyboard, since activities represent the intuitive impression of the apps in a full-screen window and are the most frequently used components for user interactions [6]. Fig. 1 shows the storyboard diagram of *Facebook* (one of most popular social media apps),

[1]"Storyboard" was developed at Walt Disney Productions, including a sequence of drawings typically with some directions and dialogues, representing the shots planned for a movie or television production.

1558-1225/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE.2019.00070

which includes the activity transition graph (ATG) with UI pages, the detailed layout code (e.g., static and dynamic), the functional code of each activity (*Activity Code*), and method call relations within each activity (*Method Hierarchy*). Based on this storyboard, PMs can review a number of apps in a short period of time and propose more competitive features in their own app.[2] UI designers can obtain the most related UI pages for reference. And developers can directly refer to the related code to improve development efficiency.

However, generating the storyboard is challenging. First, ATGs are usually incomplete due to the limitation of static analysis tools [36, 62]. Second, to identify all UI pages, a pure static approach may miss parts of UIs that are dynamically rendered (see Section III), whereas a pure dynamic approach [17, 40, 41, 66, 67] may not be able to reach all pages in the app, especially those requiring login. Third, the obfuscated activity names lack the semantics of corresponding functionalities, making the storyboard hard to understand.

To overcome these challenges, we propose a system, StoryDroid, to automatically generate the storyboard of apps in three main phases: (1) *Activity transition extraction*, which extracts ATG from the apks, especially the transitions in fragments [7] (components of activities) and inner classes [19], making ATG more complete. (2) *UI page rendering*, which first extracts the dynamic components (if any) for each UI page and embeds them into the corresponding static layout. It then renders each UI page statically based on the static layout files. (3) *Semantic name inferring*, which infers the semantic names for the obfuscated activity names by comparing the layout hierarchy with the ones in our database which is constructed from 4,426 open source apps. Through these analyses, StoryDroid provides a systematic solution for exploring and understanding an app from different points of view.

We evaluate StoryDroid on 100 apps (50 open-source and 50 closed-source apps) from the following two aspects: (1) effectiveness evaluation of each phase; (2) usefulness evaluation of the visualization outputs. The experimental results show the effectiveness of StoryDroid in extracting activity transitions, especially in fragments and inner classes. StoryDroid extracts nearly 2 times more activity transitions than the state-of-the-art ATG extraction tool (i.e., IC3 [62]) on both open-source apps and closed-source apps. Besides, StoryDroid significantly outperforms the state-of-the-art dynamic testing tool (i.e., Stoat [67]) on activity coverage for both open-source apps (87% on average) and closed-source apps (74% on average). On average, our rendered images achieve 84% similarity compared with the ones that are dynamically obtained by Stoat. And StoryDroid can infer the semantic names at a high accuracy (92%). In addition, the user study shows that with the help of StoryDroid, the activity coverage has a significant improvement compared with exploration without StoryDroid, and users can find the layout code for the given UI pages more accurately and efficiently. Apart from the fundamental usefulness, we also discuss several additional practical applications based on the outputs of StoryDroid, such as the recommendation of UI design and layout code and guiding app regression testing.

In summary, we make the following contributions:

- This is the first research work to automatically generate the storyboard of Android apps. It assists app development teams including PMs, designers, and developers to quickly have a clear overview of other similar apps.
- We propose a novel algorithm to extract relatively complete ATG, render UI pages statically, and infer activity names for obfuscated apps. These technical contributions are general for both open-source and close-source Android apps.
- Our experiments demonstrate not only the accuracy of the generated storyboard, but also the usefulness of our storyboard for assisting app development.
- This is a fundamental work to enable the construction of a large-scale database of app storyboards, as the overall approach is based on static program analysis. Such a database can expand the horizon of current mobile app research by enabling lots of future work such as extracting commonalities across apps, recommending UI code, design, and guiding app testing.

II. Motivating Scenario

We detail the typical app review process [4, 22, 34, 54, 55] with our StoryDroid for Android apps in term of different roles in the development team. Eve is a PM of an IT company. Her team plans to develop an Android social app. In order to improve the competitiveness of the designed app, she searches hundreds of similar apps (e.g., Facebook, Instagram, Twitter) based on the input keywords (e.g., social, chat) from Google Play Store. She then inputs all of the URLs of these apps into StoryDroid which automatically download all of these apps with Google Play API [2]. StoryDroid further generates the storyboard (e.g., Fig 1) of all these apps and displays them to Eve for an overview. By observing these storyboards together, she easily understands the storyline of these apps, and spots the common features among these apps such as registering, searching, setting, user profile, posting, etc. Based on these common features, Eve comes up with some unique features which can distinguish their own app from existing ones.

Alice, as a UI/UX designer, needs to design the UI pages according to Eve's requirements. With our StoryDroid, she can easily get not only a clear overview of the UI design style of related apps, but also interaction relations among different screens within the app. Then, Alice can develop the UI and user interaction of her app inspired by others' apps [16, 32].

Bob is an Android developer who needs to develop the corresponding app based on Alice's UI design. Based on Alice's referred UI design in the existing app, he can also refer to that app with the help of our StoryDroid. By clicking the UI screen of each activity in the storyboard, StoryDroid returns the corresponding UI implementation code no matter it is implemented with pure static code, dynamic code, or hybrid ones. To implement their own UI design, he can refer to the

[2]The main target is to help PMs, developers, designers understand and get inspiration from existing apps, instead of directly distributing any part of the code for developing apps for commercial purpose.

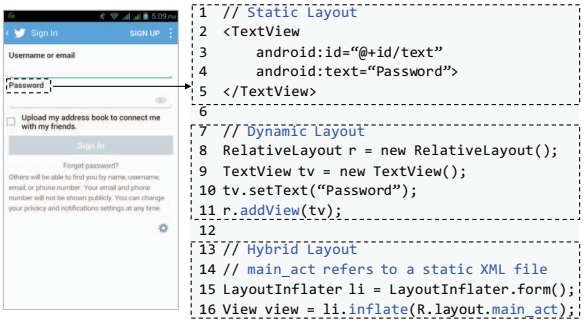


Fig. 2: Three types of GUI layout

implemented code and customize it based on their requirement. That development process is much faster than starting from scratch. In addition, Bob may also be interested in certain functionality within a certain app. By using StoryDroid, he easily locates the logic code.

III. Preliminaries

In this section, we briefly introduce the concepts of Android UI, layout types of UI and special views for populating data.

A. Android Activity and Fragment

There are four types of components in Android apps (i.e., `Activity`, `Service`, `Broadcast`, `Receiver`). Specifically, `Activity` [6] and `Fragment` [7] render the user interface and are the visible parts of Android apps. `Activity` is a fundamental component for drawing the screens which users can interact with. `Fragment` represents a portion of UIs in the activities, which contributes their own UI to certain activities. `Fragment` always depends on an `Activity` and cannot exist independently. A `Fragment` can also be reused in multiple activities and an activity may contain multiple fragments based on the screen size, with which we can create multi-panel UIs to adapt to mobile devices with different screen sizes.

B. UI Layout

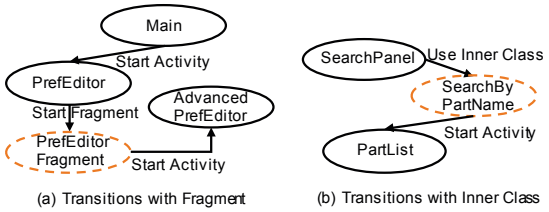
A user interface, rendered by `Activity` and `Fragment`, requires a UI layout to draw a window for users. We take a high-level look at three layout types in Android apps. Fig. 2 shows the login UI page of a famous social app, Twitter [31], where the component (e.g., `TextView`) can be implemented in three different layout types.

Static layout, which relies on the static layout files (i.e., XML) in the apk. The UI pages of the app are rendered by these XML files. `ViewGroup` and `View` are the basic elements of user interfaces in UI layout files. `ViewGroup` is a container which holds other `ViewGroups` and `Views`. The GUI code must contain a root node with `ViewGroup` (e.g., `RelativeLayout` and `LinearLayout`). When the `ViewGroup` is defined, developers can add additional `Views` (e.g., `EditText`, `TextView`) as children elements to gradually build a hierarchy that defines the page layout. GUI components contain multiply attributes (e.g., id, text, width) as shown in Fig. 2 (lines 2-5).

Dynamic layout, which allows developers to instantiate layout components at runtime using Android API (i.e., `addView()`).

```
1 public class PartList extends Activity{
2   ListView lv = (ListView) findViewById(R.id.list);
3   ArrayAdapter adapter = new
     ArrayAdapter(this,R.layout.list_view,data);
4   lv.setAdapter(adapter); ...}
```

Fig. 3: Simplified code snippet of Adapter



(a) Transitions with Fragment (b) Transitions with Inner Class

Fig. 4: Transitions between activities and fragment, inner class

Developers can create components and manipulate their attributes in Java code, e.g., TextView.setText("Password") in Java code is equivalent to android:text="Password" in layout file as shown in Fig. 2 (lines 7-11).

Hybrid layout, which defines some default components in static layout files. These layout can be reused dynamically by invoking `LayoutInflater.inflate` as shown in Figure 2 (lines 14-15). Developers can also manipulate the attributes of the defined components. For example, they can modify the content of `TextView` in Java code.

To investigate the proportion of dynamic and hybrid layout used in Android apps, we randomly select 1,000 apps from the top 10,000 Google Play apps with the most number of downloads. We use the specific Android APIs (i.e., `addView()` and `inflate()`) to distinguish if the app contains dynamic/hybrid layout types to draw the UI pages. The result of our study unveils that 62.3% apps use dynamic/hybrid layout. The reason for such frequent usage of dynamic/hybrid layout is that with the help of them, the views are separated from the view model in the XML file, developers can change the layout without recompiling to adapt to the app's runtime state.

C. Data Population

`Adapter` [5] is a bridge between the `AdapterView` and the underlying data for the view. It also provides the layout (e.g., `ListView`, `GridView`, `ViewPager`) with the data, which is usually loaded from a local database or remote server. `Adapter` enables these UI components (e.g., `ListView`) to provide a list of scrollable items for the selection purpose. Fig. 3 shows an example of `Adapter`. It instantiates an adapter with a layout (`ListView`) and associates it with `data`. The data is then displayed in a `ListView` via adapter. The `AdapterView` is part of the user interface layout and should be extracted from the source code for static rendering.

IV. Our Approach

StoryDroid takes an *apk* as input and outputs the visualized storyboard (S) for the app. Fig. 5 shows the three main phases of StoryDroid: (1) *Transition Extraction*, which enhances the ATG extraction ability of IC3 [62], especially for fragments and inner classes. StoryDroid leverages control-

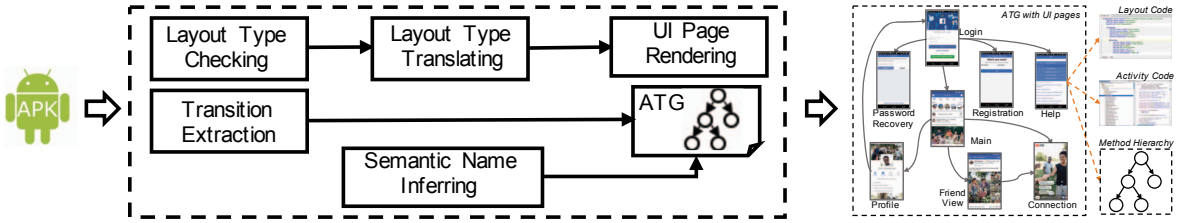


Fig. 5: Architecture of StoryDroid

```
1  public class PrefEditor{... //Using replace/add
2    PrefEditorFragment pref = new PrefEditorFragment();
3    FragmentTransaction.replace(R.id.content,pref);
4    FragmentTransaction.commit(); }
5  public class PrefEditor{... // Using setAdapter
6    ViewPager.setAdapter(getSupFragmentManager(),
7    new PrefEditorFragment()); }
```

Fig. 6: Simplified code snippet of Fragment

```
1  public class SearchPanel{...
2  private class SearchByPartName extends Asynctack<>{...
3    Intent intent = new Intent
4  (MainActivity.this,PartList.class);
5    startActivity(intent); } }
```

Fig. 7: Simplified code snippet of Inner Class

and data-flow analysis to obtain relatively complete ATG. (2) *UI Page Rendering*, which translates dynamic and hybrid layout to static layout (if needed) to render UI pages that users interact with. (3) *Semantic Name Inferring*, which infers the semantic name for the obfuscated activity names by layout comparison.

A. Transition Extraction

Before extracting activity transitions in inner classes and fragments, we illustrate the transitions in them. Fig. 4 (a) is the sub ATG of `Vespucci` [33], a map editor. Firstly, activity Main starts PrefEditor, in which PrefEditorFragment is started. And PrefEditorFragment further starts AdvancedPrefEditor. Specifically, as shown in Fig. 6, fragments can be added to an activity in two ways: (1) by invoking fragment modification API calls, e.g., "`replace()`," "`add()`," and further leveraging "`FragmentTransation.commit()`" (lines 3-4) to start the fragment; (2) By using "`setAdapter`" (line 6) to display the fragment in a certain view (e.g., `ViewPager`). The started PrefEditorFragment then starts a new activity (i.e., AdvancedPrefEditor). Fig. 4 (b) shows the sub ATG of `ADSdroid`, where SearchPanel uses an inner class SearchByPartName to handle time-consuming operations as shown in Fig. 7. After finishing the task, it starts an activity PartList by invoking "`StartActivity()`" (line 5). In this example, our goal is to extract activity transitions: Main→PreEditor, PreEditor→AdvancedPreEditor, and SearchPanel→PartList.

Algorithm 1 details the extraction of ATG and resources for rendering, including the layout type (i.e., static, dynamic, or hybrid) and the adapter mapping relations. Specifically, it takes as input an apk, and outputs the activity transition graph (atg), adapter mappings ($adapters$), and layout type ($layout\_type$). We first initialize atg as an empty set (line 1), which stores the activity transitions gradually. We then generate the call graph (cg) of the given apk, obtain the layout type by analyzing the existence of dynamic layout loading

Algorithm 1: ATG and Rendering Source Extraction

Input: apk
Output: atg: Activity transition graph, $adapters$: Adapter mappings, $layout\_type$: i.e., static, dynamic or hybrid

```
1  atg ← ∅, adapters ← ∅
2  cg ← getCallGraph(apk)
3  layout_type ← getLayoutType(apk)
4  all_classes ← getAllClasses(apk)
5  foreach c ∈ all_classes do
6      methods ← getClassMethods(c)
7      foreach m ∈ methods do
8          if hasActivityTransition(m) then
9              callee_act ← getTargetAct(m)
10             if isInnerClass(c) then
11                 caller_act ← outerClass(c)
12                 atg.addPair(caller_act, callee_act)
13             else if isInFragment(m) then
14                 atg.addPair(caller_frag, callee_act)
15             else
16                 caller_acts ← getCallerAct(m, cg)
17                 foreach act ∈ caller_act do
18                     atg.addPair(act, callee_act)
           // get rendering sources and optimize atg
19         if startFragment(m) then
20             caller_acts ← getCallerAct(caller_frag)
21             foreach act ∈ caller_acts do
22                 atg.addPair(act, callee_frag)
23             updateATGIfNeeded(atg)
           // get rendering sources
24         if hasAdapter(m) then
               // e.g., ListView, RecyclerView, ViewPager
25             view_type ← getViewType(m)
26             layout_file ← BackwardAnalysis(m)
27             adapters.add(c, view_type, layout_file)
28 return atg, adapters, layout_type
```

APIs (lines 2-3). For each method (m) in each class (c), if there exists an activity transition, we first get the target activity ($callee\_act$) by analyzing the data in `Intent` (lines 5-9). If the method (m) is in an inner class, we regard the outer class as the activity that starts the target activity and add the transition to atg (lines 10-12). Take Fig. 4 (b) as an example, we add an edge SearchPanel→PartList to atg. If m is in a fragment, we construct the relation between the fragment ($caller\_frag$) and the target activity (lines 13-14). Note that this relation does not represent the actual activity transition, we optimize it by identifying the activities that start the fragment in lines 19-22. This relation is used for both ATG construction and UI page rendering. After that we update atg by merging fragment relations to construct the actual activity transitions (line 23). For example, in Fig. 4 (a), we first obtain the relations PrefEditorFragment→AdvancedPreEditor,

PrefEditor→PrefEditorFragment, then we merge it to PrefEditor→AdvancedPreEditor to represent the actual activity transition. For method m that is neither in an inner class nor a fragment, we backward traverse cg starting from m to obtain all the activities that start the target activity ($callee\_act$), then add them to atg (lines 15-18).

In addition, to complement the UI layout that need to load data from data providers (e.g., `ContentProvider`, `Preference`) using different types of views (e.g., `ListView`, `RecyclerView`, `ViewPager`), we first identify the method that uses `Adapter` and obtain the correponding view type ($view\_type$) (lines 24-25). We then utilize backward dataflow analysis on the adapters to track the corresponding layout files, which pinpoints the layout/activity that will be embedded with data displayed in $view\_type$. We define each mapping relation as a tuple ⟨activity,view_type,layout⟩ and save them in $adatpers$ for UI rendering. Take Fig. 3 as an example, we denote the relation as ⟨PartList,ListView,list_view⟩.

B. UI Page Rendering

We propose to statically render the UI pages due to the limitation of data dependence between different activities when using dynamic tools for UI page rendering (e.g., require login or special input to reach another activity). The activities we rendered are the initial state of each activity. As for apps that only use static layout to display UI pages, we can directly extract the corresponding layout file for rendering. However, as for dynamic/hybrid layout, we need to resolve two challenges: (1) converting dynamic/hybrid layout to static layout since we cannot render the corresponding UI pages accurately with incomplete layout; (2) filling the dynamic data loading area with dummy data (i.e, text, image) since we cannot render the corresponding components that load dynamic data from remote server which involves backend code. To tackle these problems, we first statically analyze the activity source code and identify the logic that is relevant to layout population, including adding new views and modifying the parameters of views. We then convert the dynamic layouts in the target app to static layouts. Moreover, as our approach is based on static analysis, we cannot obtain the real image which is loaded dynamically such as `ListView` and `GridView`. Instead of keeping that position plain, we fill in that position with dummy images so that users can directly discriminate it from the plain background. Otherwise, the image position may be preempted by other components of the same page. We associate the dummy data with the identified adapters in Algorithm 1 to display data in different styles. With the translated static layouts and dummy dynamic data, we compile them into an apk to render UI pages.

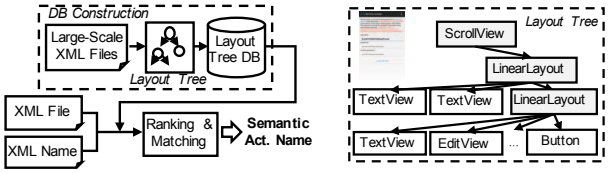
Algorithm 2 details the UI page rendering process. It takes as input the activity transition graph (atg^*), adapter mapping relations ($adapter$), layout type ($layout\_type$), and outputs the rendered UI pages. We first extract all the activities and fragments from atg^* since we need to render both activity UIs and fragment UIs to make it closer to the real UIs. For each activity/fragment (act), if act uses the static layout, we directly

Algorithm 2: UI Page Rendering

```
   Input: atg*: the transitions that contain activities and fragments,
          adapters, layout_type
   Output: pages: UI pages
1  act_frag ← getActivityAndFragment(atg*)
2  foreach act ∈ act_frag do
3      layout ← copy(act.xml)
4      if layout_type ≠ "static" then
5          par_layout ← getParentLayout(act)
6          methods ← getAllMethods(act)
7          foreach m ∈ methods do
8              if m ≠ "onCreate" then
9                  continue
10             compt, attr ← DataFlowAnalysis(m)
11             par_layout.add(compt)
               // add corresponding attributes
12             if isMethod(attr) then
13                 compt.add(DataFlowAnalysis (attr))
14             else
15                 element ← getElementByName(attr)
16                 compt.add(element)
17             break
18         act_layout.add(par_layout)
19         saveLayout(act_layout)
20 foreach adapter ∈ adapters do
21     activity ← match_activity(adapter)
22     modify_layoutFile(activity, adapter.type)
23 apk ← bulidApk(act_frag)
24 pages ← getScreenshot(apk)
25 return pages
```

make a copy of the corresponding layout file for further rendering or modification (line 3). However, if the app uses dynamic/hybrid layout, we aim to add the dynamic components together with their attributions to the corresponding parent layout. Specifically, we first extract and pinpoint the parent layout (e.g., `LinearLayout`) (line 8). We then traverse each method (m) to identify the dynamic component $compt$ (e.g., `EditText`, `TextView`) and the corresponding attributes $attr$ (e.g., text, height, width) by keywords matching (i.e., `inflate` and `addView`) and forward analyzing the data flow of view-related variables in `onCreate()`. We then add $compt$ to the corresponding parent layout (lines 7-11). As for the attribute, if it is obtained by invoking another method, we leverage dataflow analysis to reach the definition of the attribution, and attach it to the corresponding component (lines 12-13). Otherwise, we identify the corresponding element in the resource files by the attribute name via `getElementByName()`, and also attach it to $compt$ (lines 15-16). After the modification to the parent layout of the dynamic components, we attach the new par_layout to act_layout and save it as layout format for further rendering (lines 18-19). Take Fig. 2 as an example, our method is able to translate the dynamic/hybrid layouts to static layouts (i.e., XML format).

In addition, as for data display using adapters, we match each adapter to the activity and modify the corresponding layout by embedding the view types, e.g., `ListView`, `RecyclerView` (lines 20-22) as well as dummy data. We finally get a view tree and ensure the corresponding attributes are added from the source code. At last, we build an apk to render all the activities and fragments, and take screenshots



(a) Semantic name inferring (b) An example of layout tree

Fig. 8: Workflow of semantic name inferring

for each UI (lines 23-24) in a real app.

C. Semantic Name Inferring

In Android app development, activity names of apps are recommended to contain semantic meanings and end with "*Activity*" [8], thus we assume that the defined activity names by developers have basic semantic meanings of the corresponding functionality. To verify this assumption, we randomly download 1,000 apps from F-Droid [15], extract all the activity names from each app, and finally get 6,767 activity names. We manually investigate the activity names and observe that most of activity names have semantic meanings. However, Android obfuscation techniques are often used in Google Play apps to protect their security [49]. The activity names will be translated to simple words like "a," "b," and "c," totally losing their practical semantic meanings. Those obfuscated names significantly hinder users' understanding of our storyboard. To tackle this problem, we propose to automatically infer the semantic names for obfuscated activities by comparing the layout hierarchy (i.e., `ViewGroup` and `View`) with those of the existing activities since activity layout files are not obfuscated in apps. In this paper, we consider the activity names whose length is less than three letters as obfuscated ones. To infer the semantic names for obfuscated activities, we aim to learn from the activity names of open source apps.

To achieve this, as shown in Fig. 8 (a), we crawl all the apps on F-Droid (4,426 in total) and build a large database based on the layout hierarchies of the apps for similarity comparing. A layout hierarchy is a layout tree (i.e., Fig.8 (b)) based on the XML layout code. The root node of the tree is a type of `ViewGroup`, and the other `ViewGroup`s and types of `View`s are the child nodes added to the root node. We finally get 13,792 activities with their layout hierarchies. Given the XML file (layout file) and XML name of the obfuscated activity, we infer its semantic name by the following three steps: (1) *layout hierarchy extraction*, which extracts the layout tree (T) from the XML file; (2) *similarity comparison*, which computes the similarity between T and the trees in our database by leveraging the tree edit distance (TED) algorithm [70], which is defined as the minimum-cost sequence of node edit operations that transform one tree into another. According to a pilot study of 100 randomly selected tree pairs, we define a threshold in TED as 5 and filter the activity names whose TED are less than 5 as candidates. (3) *ranking and matching*, which infers the activity name from the candidates based on the frequency of each activity name and the corresponding layout names (XML file names). Specifically, we rank the activity names based on their frequency, split the camel-case layout name into multiple single words with the regular expression [48], filter out the general words such as "activity," "layout," and compare it with those in the database by keyword matching. The most matched activity name will be used to rename the obfuscated activity name. However, if the layout name is not matched with any names of the candidates, we rename it with the top frequent name. Note that although the layout names are not obfuscated, it is ineffective if we only use them to infer the semantic names since activities using dynamic layouts have no static layouts, thus have no layout names.

V. Implementation

We implement StoryDroid as an automated tool, which is written in 3K lines of Java code, and 2K lines of Python code. StoryDroid is built on top of several off-the-shelf tools: IC3, jadx [14], and Soot [28]. We use Soot to extract inputs of UI page rendering, and get the call graphs from apks. Activity transition extraction is built on IC3 to obtain a comparatively complete ATG. jadx is used to decompile the apk to the source code for Android apps. We also use jadx to extract XML layout code from each apk, which will not be affected by obfuscation, thus will not affect the semantic name inferring method. We use data-driven document (D3) [13] to visualize StoryDroid's results, which provides a visualized technique based on data in `HTML`, `JavaScript`, and `CSS`. The visualization [25] contains 4 parts: (1) ATG with activity names and corresponding UI pages; (2) The layout code of each UI page; (3) The functional code of each activity; and (4) the method call relations within each activity.

VI. Effectiveness Evaluation

In this section, we evaluate the effectiveness of StoryDroid based on the following three research questions:

RQ1: Can StoryDroid extract a more complete ATG for an app, and achieve better activity coverage than the dynamic testing tool (i.e., Stoat)?

RQ2: Can StoryDroid render UI pages with high similarity compared with the real screenshots?

RQ3: Can StoryDroid infer accurate semantic names for obfuscated activities?

A. Experimental Setup

To investigate the capability of handling fragments and inner classes, we self-developed 10 apps[3] as our ground-truth benchmark, which cover different features (i.e., activity, fragment, and inner class) we claim to resolve. This is the only way to exactly know all the transitions even in fragments or inner classes, and this way is widely used in the literature [57]. We follow the features below to generate the apps: (1) apps with transitions only in activities; apps with transitions only in inner classes; apps with transitions only in fragments; apps with transitions in both activities and inner classes; apps with transitions in both activities and fragments; (2) we developed 2 apps with 7-10 activities under each rule. We apply both IC3 and StoryDroid on the 10 apps and extract

[3]The 10 apps are available on https://sites.google.com/view/storydroid/

TABLE I: Capability of handling fragments and inner classes. ("-": IC3 cannot handle)

App ID	Feature	#Transition Pairs	#Identified by IC3	#Identified by StoryDoid
1	Activity	14	14	14
2		13	13	13
3	Inner Class	13	-	13
4		13	-	13
5	Fragment	13	-	13
6		13	-	13
7	Activity Inner Class	13	1	13
8		13	1	13
9	Activity Fragment	10	1	10
10		10	1	10

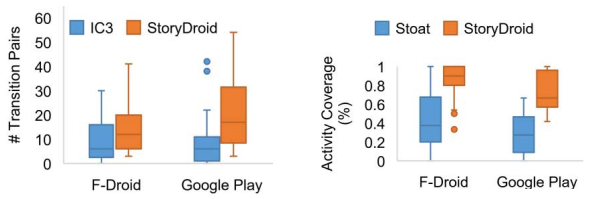
the transition pairs respectively. Moreover, since the stakeholders are more concerned about popular competitive apps which have already dominated the market. To mimic the real scenario, we randomly download 50 apps from GooglePlay Store with 10M+ installations to demonstrate the effectiveness of StoryDroid on real-world apps. We compare the number of transition pairs identified by IC3 and StoryDroid. We also use these 100 apps to evaluate the activity coverage of StoryDroid and the state-of-the-art dynamic testing tool, Stoat [67], which has been demonstrated to be more effective on app exploration than other tools such as Monkey [17] and Sapienz [58]. Specifically, we collect all the activities defined in each app from `AndroidManifest.xml`, and compare the number of the rendered activities by StoryDroid and the explored activities by Stoat.

For RQ2, we evaluate the similarity of our statically rendered UI pages with the real UI which are dynamically rendered UI by Stoat based on the 100 apps [40]. Stoat is configured with default settings and given 30 minutes to test each app in order to collect the explored activities. We further apply Stoat on each app to collect the screenshots of each activity. Since Stoat may only explore part of the activities of an app within the given time, we only compare the similarity of the explored activities with the corresponding rendered activities by StoryDroid for a fair comparison.

For RQ3, to demonstrate the accuracy of inferring semantic names, we randomly select 100 activity names with semantic meanings from the extracted 6,767 activity names in Section IV-C as the ground truth. We collect the corresponding layout files from source files, and further utilize our method based on TED [70] to obtain the semantic names for the 100 activities based on our collection of layout trees (i.e., 13,792 layout files). We then compare the results with the original activity names to evaluate the accuracy of our approach i.e., the proportion of semantic names that are correctly inferred.

B. Experimental Results

1) RQ1: Table I shows the results of the capability evaluation of handling fragments and inner classes on the 10 ground-truth benchmark apps. We can see that IC3 is able to extract transitions in activities, however, is weak in fragments and inner classes. In contrast, StoryDroid can extract transitions with respect to all these features. Since we extract transitions by using particular APIs (i.e., `StartActivity`, `StartActivityForReulst`, `StrartActivityIfNeeded`) that start new activities with data-flow analysis, the extracted transitions are more accurate. The results demonstrate the effectiveness of StoryDroid on extracting activity transitions, especially in fragments and inner classes. Moreover, we also evaluate the effectiveness of StoryDroid on real apps compared with IC3 in Fig 9 (a). It shows that StoryDroid extracts nearly 2 times more activity transitions and is more effective than IC3 for both open-source apps and closed-source apps.



(a) Comparison with IC3 (b) Comparison with Stoat

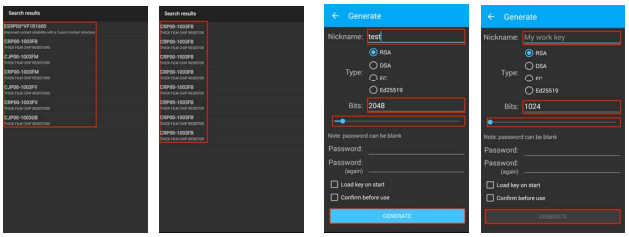
Fig. 9: Comparison of transitions pairs and activity coverage

Besides the limitations of inner classes and fragments, transitions in Android system event callbacks are lost in IC3 according to our observation. For example, (1) `PodListen` [26] is a podcast player, and there exists an activity transition in the callback method (`DownloadReceiver.onReceive()`), which is called when the `BroadcastReceiver` is receiving an `Intent` broadcast. However, IC3 fails to extract the activity transition. (2) `CSipSimple` [12] is an online voice communication app, in which the system callback method (`SipService.adjustVolume()`) starts a new activity, but IC3 fails to identify the transition.

Fig. 9 (b) depicts the activity coverage results. On average, StoryDroid outperforms Stoat in terms of activity coverage, achieving 87% and 74% coverage on open-source apps and closed-source apps, respectively. In addition, StoryDroid costs much less time (i.e., within 3 minutes on average) to extract and render the activities than Stoat (i.e., 30 minutes). StoryDroid does not cover all the activities for some apps due to the following reasons: (1) the limitation of reverse engineering techniques, some classes and methods cannot be decompiled from the apks, causing failure in the extraction of activity transition and coverage. That situation is more severe in closed source apps due to packing [9] and obfuscation. In our evaluated apps, `PodListen` is the only project in which activity coverage of StoryDroid is lower than that of Stoat due to decompiling failure of `SubscribeDialog` fragment. (2) Another reason is the dead activities (no transitions), such as unused legacy code and testing code in apps.

We further investigate the reasons why Stoat achieves low activity coverage: (1) Login requirement. For example, Stoat fails to explore `Santander` which is a banking app requiring login using password or fingerprint. (2) Lack of specific events. For example, `Open Training` is a fitness-training app, which can create fitness plans by swiping across the screen. However, Stoat does not support such events, resulting low coverage.

2) RQ2: As for the similarity of the rendered images compared with the ones obtained by Stoat, we use two widely used image similarity metrics [3] (i.e., mean absolute



(a) Real page (b) Our rendered (c) Real page (d) Our rendered

Fig. 10: Comparison of real pages and rendered pages

TABLE II: Partial results of inferring semantic name for obfuscated activities

GroundTruth Act. Name	Rank in Candidates	Corresponding XML Name	Inferred by StoryDroid
AboutAct.	1	about	AboutAct.
HelpAct.	2	activity_help	HelpAct.
PersonalInfoAct.	3	content_extended_title	**WizardAct.**
LoginAct.	3	login	LoginAct.
ContactListAct.	1	contact_list	ContactListAct.
SearchAct.	4	grid_base	**Searcher**
SettingAct.	1	setting_container	SettingAct.
ShareAct.	1	activity_share	ShareAct.
SplashAct.	3	activity_splash	SplashAct.
TrackListAct.	1	list_view	**TrackListAct.**

error (MAE) and mean squared error (MSE)) to measure the similarity pixel by pixel. MAE measures the average magnitude of differences between the prediction and the actual value and MSE measures the average squared differences between them. On average, our rendered images achieve 84% and 88% similarity in terms of MAE and MSE, respectively. The reasons for the inconsistencies are explained as follows. (1) Some data in components are dynamically loaded from web servers or local storage (e.g., `Preference` and SD card), such as the list data of `ListView`, the text data of `TextView`, and the image background of `ImageView`. The differences are depicted in Fig. 10 (a), (b). (2) Some components (e.g., `Button`) will change their color or visibility after users type some data. For example, as shown in Fig. 10 (c), the color of the button changes when users input the "Nickname". StoryDroid renders the initial state of the activity, thus reducing the similarity of the two UI pages, which however does not affect the understanding of the app functionality.

3) RQ3: 92 out of 100 activity names are inferred correctly based on our approach, and the accuracy is 92%. To demonstrate and explain the inferring results, we randomly select 10 cases listed in Table II. The first column refers to the ground-truth activity names. According to the results of comparison with layout trees in our database, we list the ranks of correct activity names based on the frequency of each name in the second column. The third column is the corresponding XML layout names. StoryDroid accurately infers 8 out of the 10 semantic names which are consistent with the ground truth. For `TrackListAct.`, `PersonalInfoAct.` and `SearchAct.` (highlighted in gray color in Table II), the layout names cannot be matched with any name of the candidates, so we choose the name at rank one as the semantic name. Two of them do not match the ground truth. But note that the performance of our approach is highly underestimated, as although some of the recommended names from our StoryDroid differ from the ground truth, they actually have similar meanings, such as `SearchAct.` and `Searcher`. In addition, with the expansion of our database, the accuracy will be also boosted. Overall, the results show that StoryDroid can help infer the semantic names of obfuscated activities effectively.

Remark: StoryDroid outperforms IC3 on ATG extraction and covers 2 times more activities than Stoat with less time. StoryDroid can render UI pages with high similarity (84%) to the real ones and accurately (92%) infer the semantic names for obfuscated activity names.

VII. Usefulness Evaluation

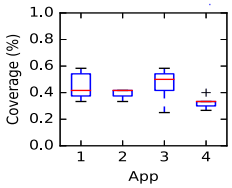
Apart from effectiveness evaluations, we further conduct a user study to demonstrate the usefulness of StoryDroid. Our goals are to check: (1) whether StoryDroid can help explore and understand the functionalities of apps effectively? (2) whether StoryDroid can help identify the layout code of the given UI page accurately and effectively?

Dataset of user study. We randomly select 4 apps (i.e., `Bitcoin`, `Bankdroid`, `ConnectBot`, `Vespucci`) with different number of activities (12-15 activities) from 2 categories (i.e., finance, tool), which are hosted on Google Play Store. Each category contains two apps, and we ask participants to explore each app to finish the assigned tasks.

Participant recruitment. We recruit 8 people including post-doc, Ph.D, and master from our university to participate in the experiment via word-of-mouth. All of the recruited participants have used Android devices for more than one year, and participated in Android related research topics. They never use these apps before. All of them have Android app development experience and come from different countries, such as USA, China, European countries (e.g., Spain), and Singapore. The participants receive a $10 shopping coupon as a compensation of their time.

Experiment procedures. We installed the 4 apps on an Android device (Nexus 5 with Android 4.4). The experiment started with a brief introduction to the tasks. We explained and went through all the features we want them to use within the apps and asked each participant to explore the 4 apps separately to finish the tasks below. Note that for each category, each participant explored one app with StoryDroid, and the other without StoryDroid. To avoid potential bias, the order of app category, and the order of using StoryDroid or not using are rotated based on the Latin Square [69]. This setup ensures that each app is explored by multiple participants with different development experience. We told each participant to complete two tasks with the given apps: (1) manually explore as many functionalities of the apps as possible in 10 minutes, which is far longer than the typical average app session (71.56 seconds) [39], and understand the app functionalities with StoryDroid; (2) identify the corresponding layout code for the given 2 UI pages in 10 minutes. The 2 UI pages are implemented by static and dynamic layout types respectively. After the exploration, participants were asked to rate their satisfactoriness in exploration and confidence in mapping UI

TABLE III: User study results of app exploration. The figure represents the activity coverage of the 4 apps with manual exploration. $*$ denotes $p < 0.01$ and $**$ denotes $p < 0.05$.



Metrics	Manual Exploration	Story -Droid
Time (min)	5.2	2.5*
Coverage	40.8%	86.5%*
Satisfactoriness (1-5)	4.2	4.4**

page and code (on the 5-point likert scale with 1 being least satisfied or confident and 5 being most satisfied or confident). All participants carried out experiments independently without any discussions with each other. After performing all tasks, they were required to write some comments about our tool.

Experiment results. As displayed in Table III, the average activity coverage of manual exploration is quite low (i.e., 40.8%), showing the difficulty in exploring app functionalities thoroughly by manual exploration. However, the participants' satisfactoriness of completeness of exploration is high (i.e., 4.2 on average). It indicates that the development teams sometimes are not aware that they miss many features when exploring others' apps. Such blind confidence and neglection may further negatively influence their strategy or decision in developing their own apps. Compared with manual exploration, StoryDroid achieves 2 times more activity coverage with less time cost (2.5 minutes on average) to help understand the app functionalities. According to the participates' feedback, the average satisfactoriness of StoryDroid is 4.4, which represents the usefulness of helping participants explore and understand app functionalities. Table IV shows the time cost and confidence of mapping UI page to layout code with and without StoryDroid. All the participants spend over 8 minutes to map the UI pages to the corresponding layout code, among which 4 pages are not mapped successfully within 10 minutes. In contrast, with the help of StoryDroid, participants only spent 30 seconds to confirm the mapping relations between UI pages and layout code. To understand the significance of the differences between without StoryDroid and with StoryDroid, we carry out the Mann-Whitney U test [21], which is designed for small samples. The results in Table III and Table IV are both significant with p-value $<$ 0.01 or p-value < 0.05.

We analyze the comments from the participants, and find that they mainly focus on two aspects: (1) It is better to add activity transition methods/events on the edges, such as clicking "login"; (2) If the transition graph is too complex, StoryDroid needs to provide a better strategy to visualize it.

VIII. Discussion

A. Future Applications Based on StoryDroid

Apart from the above fundamental usefulness (i.e., exploration of app functionalities), we discuss additional follow-up applications based on the outputs of StoryDroid.

Recommendation of UI design and code. Developing the GUI of a mobile application involves two steps, i.e., UI design and implementation. Designing a UI focuses on proper user interaction and visual effects, while implementing a UI focuses on making the UI work as designed with proper layouts and widgets of a GUI framework. Our StoryDroid can assist both UI designers and developers by building a large-scale database of app storyboard. Such a database bridges the gap across the abstract activities (text), UI pages (image) and detailed layout code (i.e., activity→UI page→layout code) so that they can be searched as a whole. Due to that mapping, UI/UX designers can directly use keywords (e.g., "Login" and "Search") to search the UI images by matching the activity name of the UI in our database. The searched images can be used for inspiring their own UI design. The UI developers can also benefit from searching our database for UI implementation. Given the UI design image from designers, developers can search the similar UI in our database by computing image similarity (e.g., MSE in Section VI-B3). As each UI page in our database is also associated with corresponding run-time UI code, developers can select the most related UI page in the candidate list and then customize the UI code for their own need to implement the given UI design.

TABLE IV: User study results of UI page and layout code mapping. $*$ denotes $p < 0.01$ and $**$ denotes $p < 0.05$.

Metrics	Without StoryDroid	With StoryDroid
Time Cost (min)	8.5	0.5*
Confidence (1-5)	4.5	5.0**

Guiding regression testing of apps. Reusing test cases is useful to improve the efficiency of regression testing for Android apps [65]. StoryDroid can help guide app regression testing by identifying the UI components that have been modified. Different versions of a single app have many common functionalities, which means most of UI pages in the newer version are the same as the previous version. StoryDroid stores the mapping relation between UI page and the corresponding layout code, therefore, analyzers can obtain the modified components by analyzing the differences of layout code and ATG, and further update the corresponding test cases. In this scenario, most of the test cases can be reused, and the modified components can be identified effectively to guide test case update for regression testing.

B. Limitations

Transition Extraction. The inputs of UI page rendering are extracted from static analysis based on Soot, but some files failed to be transformed, and the call graphs can still be incomplete. As for the closed-source apps, jadx is used to decompile apk to Java code. However, some Java files failed to be decompiled, which affects the analysis results of UI page rendering. But according to our observation, these cases rarely appear in the real apps. Besides, as the activities spawned by other components (e.g., Broadcast Receiver) can only be dynamically loaded, our static-analysis based approach cannot deal with them.

UI Page Rendering. (1) Some UI pages use self-defined components by overwriting the method `onDraw`. For example,

Kiwix [20] lets users read Wikipedia without Internet connections. It draws the canvas for the UI page dynamically. (2) Some components are dynamically created from a user-defined function. For example, BankDroid [10] is a banking client for the Swedish banks. It invokes CreateFrom() function to create a component where the parameter is the return value of another self-defined method. As we cannot get a clue of what these values are unless we analyze them at runtime, we use some empty placeholder to take the position so that the overall layout is the same as the real UI page.

IX. Related Work

Studies for helping Android development. GUI provides a visual bridge between apps and users through which they can interact with each other. Developing the GUI of a mobile app involves two separate but related activities: design the UI and implement the UI. To assist UI implementation, Nguyen and Csallner [61] reverse-engineer the UI screenshots by image processing techniques. More powerful deep-learning based algorithms [38, 40, 59] are further proposed to leverage the existing big data of Android apps. Retrieval-based methods [37, 64] are also used to develop the user interfaces. Reiss [64] parses the sketch into structured queries to search related UIs of Java-based desktop software in the database.

Different from the UI implementation studies, our study focuses more on the generation of app storyboard which not only contains the UI code, but also the transitions among the UIs. In addition, the UI code generated in prior work [38, 40, 59, 61] is all static layout, which conflicts with our observation in Section III that developers often write Java code to dynamically render the UI. In our work, we provide developers with the original UI code (no matter static code, dynamic code, or hybrid) for each screen. Such real code makes developers more easy to customize the UIs for their own needs. Apart from the UI implementation, some studies also explore issues between UI design and its implementation. Moran et al [60] check whether the UI implementation violates the original UI design by comparing the image similarity with computer vision techniques. They further detect and summarize GUI changes in evolving mobile apps. They rely on the dynamically running apps for collecting UI screenshots, and that is time-consuming and leads to low coverage of the app. In contrast, our method can extract most UI pages of the app statically, so it can complement with these studies for related tasks.

GUIFECTCH [37] customizes Reiss's method [64] into Android app UI search by considering the transitions between UIs. It can also extract UI screenshots with corresponding transitions, but our work is different from theirs in two aspects. First, their model can only deal with open-source apps, while ours can also reverse-engineer the closed-source apps, hence leading to more generality and flexibility. On the other hand, GUIFECTCH is much more heavy-weight than our static-analysis based approach, as it relies on both static analysis for UI code extraction and dynamic analysis for transition extraction. In addition, dynamically running the app usually cannot cover all screens, leading to the loss of information.

Studies for helping app understanding. The process of reverse engineering of Android apps is that researchers rely on the state-of-the-art tools (e.g., APKTOOL [29], ANDROGUARD [27], DEX2JAR [30], SOOT [28]) for decompiling an APK to intermediate language (e.g., smali, jimple) or Java code. Android reverse engineering is usually used to understand and analyze apps [1]. It also can be used to extract features for Android malware detection [46]. However, reverse engineering only has the basic functionality for code review. Different from reverse engineering, our work provides a storyboard of each app to show the basic functionalities and other useful mappings between the UI page and the corresponding layout code, which helps different parties (e.g., PM, UI/UX designer, developer) improve their work efficiency in the real world.

Studies for analyzing Android apps. Many static analysis techniques [35, 36, 42, 43, 50, 51, 56, 62, 63] have been proposed for Android apps. A^{3}E provides two strategies, targeted and depth-first exploration, for systematic testing of Android apps [36]. It also extracts static activity transition graphs for automatically generated test cases. Apart from the target of Android testing, we extract activity transition graphs to identify and systematically explore the storyboard of Android apps. EPICC is the first work to extract component communication [63], and it determines most Intent attributes to component matching. ICC [62] significantly outperforms EPICC on the extraction ability of inter-component communication by utilizing the solver for MVC problems based on the proposed COAL lauguage. FLOWDROID [35] and ICCTA [56] extract call graphs based on SOOT for data-flow analysis for detecting data leakage and malicious behaviors [44–47, 52, 68].

X. Conclusion and Future work

In this paper, we propose STORYDROID, a system to return visualized storyboard of Android apps by extracting relatively complete ATG, rendering the UI pages statically, and inferring semantic name for obfuscated activities. Such a storyboard benefits different roles (i.e., PMs, UI designers, and developers) in the app development process. The extensive experiment and user study demonstrate the effectiveness and usefulness of STORYDROID. Based on the outputs of STORYDROID, we are able to construct a large-scale database of storyboard to bridge the gap across app activities (text), UI pages (image), and implementation code. Such a comprehensive database can enable many potential applications such as recommending UI pages to designers and implementation code for developers. In the future, we will explore these potential applications, and also extend our approach into other platforms such as IOS apps and desktop software for more general usage.

Acknowledgement

We appreciate the constructive comments from Prof. Zhenchang Xing and Li Li. This work is partially supported by NSFC Grant 61502170, the Science and Technology Commission of Shanghai Municipality Grants 18511103802 and 18511103802, NTU Research Grant NGF-2017-03-033 and NRF Grant CRDCG2017-S04.

REFERENCES

[1] (2013) How to analyze APK and understand it. [Online]. Available: https://reverseengineering.stackexchange.com/questions/2703/how-do-i-analyze-a-apk-file-and-understand-its-working

[2] (2014) Crawl and download apps from Google Play. [Online]. Available: https://github.com/dflower/google-play-crawler

[3] (2016) Mean Absolute Error and Mean Squared Error. [Online]. Available: https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

[4] (2018) 4 steps to develop your app idea. [Online]. Available: http://apptology.com/blog/tag/mobile-app-storyboard/

[5] (2018) Android Adapter. [Online]. Available: https://developer.android.com/reference/android/widget/Adapter

[6] (2018) Android documentation: Activity. [Online]. Available: https://developer.android.com/reference/android/app/Activity

[7] (2018) Android Fragment. [Online]. Available: https://developer.android.com/reference/android/app/Fragment

[8] (2018) Android naming conventions. [Online]. Available: https://medium.com/@mikelimantara/overview-of-android-project-structure-and-naming-conventions-b08f6d0b7291

[9] (2018) Android Packer Tehchniques. [Online]. Available: http://www.ninoishere.com/android-packer/

[10] (2018) Bankdroid. [Online]. Available: https://play.google.com/store/apps/details?id=com.liato.bankdroid

[11] (2018) Competitor analysis before launching a mobile app startup. [Online]. Available: https://growthbug.com/competitor-analysis-before-launching-a-mobile-app-startup-f2f6a19f21b7

[12] (2018) Csipsimple. [Online]. Available: https://github.com/tqcenglish/CSipSimple

[13] (2018) D3.js. [Online]. Available: https://d3js.org/

[14] (2018) Dex to Java decompiler. [Online]. Available: https://github.com/skylot/jadx

[15] (2018) F-droid Market. [Online]. Available: https://f-droid.org/en/packages/

[16] (2018) Getting better at design is easy, just copy people! [Online]. Available: https://medium.com/ux-power-tools/getting-better-at-design-is-easy-just-copy-people-f19ba3be8a62

[17] (2018) Google Monkey for Testing. [Online]. Available: https://developer.android.com/studio/test/monkey

[18] (2018) How Much Does an App Cost. [Online]. Available: https://savvyapps.com/blog/how-much-does-app-cost-massive-review-pricing-budget-considerations

[19] (2018) Java Inner Class. [Online]. Available: https://www.tutorialspoint.com/java/java_innerclasses.htm

[20] (2018) Kiwix. [Online]. Available: https://play.google.com/store/apps/details?id=org.kiwix.kiwixmobile

[21] (2018) Mann-Whitney U test. [Online]. Available: http://www.statisticssolutions.com/mann-whitney-u-test/

[22] (2018) Mobile app development process. [Online]. Available: https://thebhwgroup.com/blog/mobile-app-development-process

[23] (2018) Mobile Internet use passes desktop for the first time. [Online]. Available: https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/

[24] (2018) Number of apps available in leading app stores as of 1st quarter. [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[25] (2018) Overview of StoryDroid. [Online]. Available: https://sites.google.com/view/storydroid/

[26] (2018) Podlisten. [Online]. Available: https://f-droid.org/en/packages/com.einmalfel.podlisten/

[27] (2018) Reverse engineering of Android applications. [Online]. Available: https://github.com/androguard/androguard

[28] (2018) Soot: A Java optimization framework. [Online]. Available: https://github.com/Sable/soot

[29] (2018) A tool for reverse engineering Android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[30] (2018) Tools to work with Android .dex and Java .class files. [Online]. Available: https://github.com/pxb1988/dex2jar

[31] (2018) Twitter. [Online]. Available: https://play.google.com/store/apps/details?id=com.twitter.android

[32] (2018) Uninvited Redesigns. [Online]. Available: https://uninvitedredesigns.com/

[33] (2018) Vespucci. [Online]. Available: https://play.google.com/store/apps/details?id=de.blau.android

[34] J. J. Arbon, *App quality: Secrets for agile app teams*. Jason Arbon, 2014.

[35] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[36] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 641–660.

[37] F. Behrang, S. P. Reiss, and A. Orso, "Guifetch: Supporting app design and development through GUI search," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 2018, pp. 236–246.

[38] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM, 2018, p. 3.

[39] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage," in *Proceedings of the 13th international conference on Human computer interaction with mobile devices and services*. ACM, 2011, pp. 47–56.

[40] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 665–676.

[41] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu, "Automated cross-platform GUI code generation for mobile apps," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE, 2019.

[42] S. Chen, G. Meng, T. Su, L. Fan, Y. Xue, Y. Liu, L. Xu, M. Xue, B. Li, and S. Hao, "AUSERA: Large-scale automated security risk assessment of global mobile banking apps," *arXiv preprint arXiv:1805.05236*, 2018.

[43] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? What can be improved?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 797–802.

[44] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *computers & security*, vol. 73, pp. 326–344, 2018.

[45] S. Chen, M. Xue, L. Fan, L. Ma, Y. Liu, and L. Xu, "How can we craft large-scale Android malware? An automated poisoning attack," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE, 2019.

[46] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting Android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIACCS*. ACM, 2016, pp. 377–388.

[47] S. Chen, M. Xue, and L. Xu, "Towards adversarial detection of mobile malware: poster," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. ACM, 2016, pp. 415–416.

[48] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *2011 19th IEEE International Conference on Program Comprehension*. IEEE, 2011, pp. 11–20.

[49] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," *arXiv preprint arXiv:1801.01633*, 2018.

[50] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in Android apps," in *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE, Montpellier, France, May 27 - June 03*, 2018, pp. 485–496.

[51] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in Android apps," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 408–419.

[52] L. Fan, M. Xue, S. Chen, L. Xu, and H. Zhu, "Poster: Accuracy vs. time cost: Detecting Android malware through pareto ensemble pruning," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1748–1750.

[53] C. Finch and P. Blake, *The art of Walt Disney: From Mickey mouse to the magic kingdoms*. Abrams, 1995.

[54] R. Fox, "Mobile app development: The effect of smartphones, mobile applications and geolocation services on the tourist experience," Ph.D. dissertation, University of Baltimore, 2017.

[55] L. Guo, R. Sharma, L. Yin, R. Lu, and K. Rong, "Automated competitor analysis using big data analytics: Evidence from the fitness mobile app business," *Business Process Management Journal*, vol. 23, no. 3, pp. 735–762, 2017.

[56] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detectingz inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 280–291.

[57] L. Li, A. Bartel, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel, "I know what leaked in your pocket: Uncovering privacy leaks on Android apps with static taint analysis," *arXiv preprint arXiv:1404.7431*, 2014.

[58] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 94–105.

[59] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *arXiv preprint arXiv:1802.02312*, 2018.

[60] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of GUI design violations for mobile apps," *arXiv preprint arXiv:1802.04732*, 2018.

[61] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 248–259.

[62] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 77–88.

[63] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in Android with epicc: An essential step towards holistic security analysis," *Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis*, 2013.

[64] S. P. Reiss, Y. Miao, and Q. Xin, "Seeking the user interface," *Automated Software Engineering*, pp. 157–193, 2018.

[65] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[66] T. Su, "Fsmdroid: guided GUI testing of Android apps," in *IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 689–691.

[67] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.

[68] C. Tang, S. Chen, L. Fan, L. Xu, Y. Liu, Z. Tang, and L. Dou, "A large-scale empirical study on industrial fake apps," in *Proceedings of the 41th ACM/IEEE International Conference on Software Engineering, ICSE*. IEEE, 2019.

[69] B. J. Winer, "Statistical principles in experimental design." 1962.

[70] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.