Automatically Distilling Storyboard with Rich Features for Android Apps

Sen Chen, Lingling Fan, Chunyang Chen, and Yang Liu

Abstract—Before developing a new mobile 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 inspirations for app development. However, manual exploration of hundreds of existing apps by different roles (e.g., product manager, UI/UX designer, developer, and tester) can be ineffective. For example, it is difficult to completely explore all the functionalities of the app from different aspects including design, implementation, and testing in a short period of time. However, existing reverse engineering tools only provide basic features such as AndroidManifest.xml and Java source files for users.

Following the conception of storyboard in movie production, we propose a system, named StoryDistiller, to automatically generate the storyboards for Android apps with rich features through reverse engineering, and assist different roles to review and analyze apps effectively and efficiently. Specifically, we (1) propose a hybrid method to extract a relatively complete Activity transition graph (ATG), that is, it first extracts the ATG of Android apps through static analysis method first, and further leverages dynamic component exploration to augment ATG; (2) extract the required inter-component communication (ICC) data of each target Activity by leveraging static data-flow analysis and renders UI pages dynamically by using app instrumentation together with the extracted required ICC data; (3) obtain rich features including comprehensive ATG with rendered UI pages, semantic activity names, corresponding logic and layout code, etc. (4) implement the storyboard visualization as a web service with the rendered UI pages and the corresponding rich features. Our experiments unveil that StoryDistiller is effective and indeed useful to assist app exploration and review. We also conduct a comprehensive comparison study to demonstrate better performance over IC3, Gator, Stoat, and StoryDroid.

Index Terms—Android apps, app review, competitive analysis, reverse engineering, storyboard

1 Introduction

Mobile applications (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 [2]. 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 [3]. Additionally, a large number of mobile apps still suffer from functional bugs [4], [5], security vulnerabilities [6], [7], [8], and the lack of marketing competitiveness.

Therefore, for app developers and companies, it is crucial to perform extensive competitive analysis through app review over existing apps with similar purposes [9], [10], [11], [12]. 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 insights on the actual implementation, given that delivering commercial apps can be time-consuming and expensive [13]. Besides, from the perspective of app testers for testing purpose, they aim to catch more useful features, such as logic, functionalities, and version changes. However, to the best of our knowledge, existing reverse engineering tools can only provide partial features, such as the configuration file (i.e., AndroidManifest.xml) and Java source files to analysts directly [14].

To achieve the aforementioned tasks such as competitive 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 he is interested in [9], [10], [12], [11]. However, such manual exploration can be painstaking and ineffective. For example, if a tech company plans to develop a social media app, over 200 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 be manually uncovered all functionalities in a reasonable time [15]. 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 layout

- *Sen Chen is with the College of Intelligence and Computing, Tianjin University, China. Email: senchen@tju.edu.cn. Lingling Fan (Corresponding author) is with the College of Cyber Science, Nankai University, China. Email: linglingfan@nankai.edu.cn. Chunyang Chen is with the Faculty of Information Technology, Monash University, Australia. Email: chunyang.chen@monash.edu. Yang Liu is with the Zhejiang Sci-Tech University, China and School of Computer Science and Engineering, Nanyang Technological University. Email: yangliu@ntu.edu.sg.*
- *This work is an extension to our previous paper published in ICSE'19 [1]. This journal version has substantially extended our conference version in terms of technique contributions and experiments.*

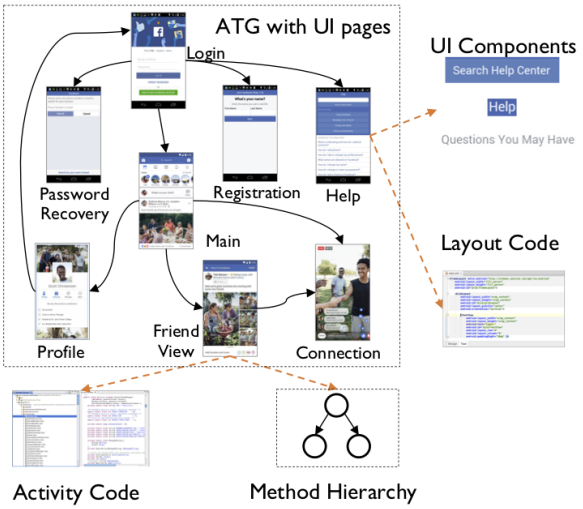


Fig. 1: The storyboard diagram of an Android app

files as well as a large piece of functional code. For app testers who want to understand the existing apps in depth from multiple aspects, such as app logic, functionalities, and version changes in order to design test cases or testing strategies, it is difficult to obtain all the useful features at the same time with existing reverse engineering tools, such as ApkTool [16] and Androguard [17].

Inspired by the conception of ***storyboard***[1] in movie industry [18], we intend to generate the storyboard of an Android app to visualize its key app behaviors and rich features. 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 [19]. Fig. 1 shows the storyboard diagram of *Facebook* (one of the most popular social media apps), which includes the activity transition graph (ATG) with UI pages, the detailed layout code, independent UI components, 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. Meanwhile, app testers can understand the main logic, functionalities, as well as version changes to generate test cases.

However, generating storyboards is challenging. First, ATG is usually incomplete with low activity coverage due to the limitations of static analysis tools such as A3E [15], IC3 [20], and Gator [21]. Second, to render all UI pages, a pure static approach may miss parts of UIs that are dynamically rendered and reduce UI similarity compared with real pages, whereas existing pure dynamic approaches [22], [23], [24], [25] can only reach limited activities in the app, especially for those requiring login. Third, the obfuscated activity names lack the semantics of corresponding functionalities, making the storyboard hard to understand.

In our previous conference version [1], to overcome these challenges, we propose a system (named **StoryDroid**) to automatically generate the storyboards of apps in three main phases: (1) *Activity transition extraction*, which extracts ATG from the apks, especially the transitions in fragments [26] (components of Activity) and inner classes [27], making ATG more complete. (2) *Static 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.[3]

However, there are still some limitations in StoryDroid [1], which motivates us to extend to this journal version. (1) The completeness of ATG is still not satisfying (below 70% activity coverage on average) especially for the closed-source apps (below 60%) due to the limitations of pure static analysis such as decompilation errors and dynamic-loading components. (2) Some of the rendered pages by the pure static method have a big visual difference compared with the real pages (Fig. 9 (a)) even though it achieves ~80% similarity on average. More importantly, not all the dynamic/hybrid layout code can be transferred to static layout code, causing unexpected errors such as rendering failures of user-defined components, third-party dependency errors, and resource file errors, which directly leads to low success rate of page rendering (~55% launch ratio on average). These above issues significantly reduce the usability of storyboards in practice.

However, it is non-trivial to overcome the above limitations, because it is challenging to further improve (1) the completeness of ATG only by a pure static method because it is hard to handle various types of activity startups or address the limitations caused by code reverse engineering [20], [21], [1], [15]; (2) the capability of static UI page rendering because it cannot transfer various types of dynamic components and is hard to render UI pages of closed-source apps due to compilation failures. To address the limitations of the pure static method in StoryDroid [1], we propose a *hybrid approach* named **StoryDistiller**, which combines static and dynamic methods to distill and generate storyboards for Android apps more effectively, and further help different stakeholders to explore and review apps. Consequently, in this paper, we make substantial effort to upgrade the generation capability of storyboards for apps from the following technical aspects:

- In terms of the *Activity transition extraction*, we leverage *Dynamic UI component exploration* to dynamically augment the transition graph extracted by the pure static method in StoryDroid. Consequently, StoryDistiller combines the

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.

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

3. According to a pilot study on 1,000 randomly selected activities names, we found that *few activity names* lack semantics in the experimental dataset. Therefore, in this version, to make the paper more compact, we did not pay more attention to the *semantic name inferring*.

advantages of static and dynamic methods with over 20% increase in activity transition pairs and more than 10% improvement in activity coverage.
- As for the *Dynamic UI page rendering*, we leverage static data-flow analysis to extract the inter-component communication (ICC) data transferred across different activities. Based on it, StoryDistiller can render UI pages dynamically with a high success launch ratio (**~80%** vs. ~55% in StoryDroid on average) and can address the low page similarity of the static rendering method[4] used in StoryDroid (**~95%** vs. ~80% on average).
- StoryDistiller provides a web service to visualize the storyboards with rich features and enhance the usability of StoryDistiller. Thanks to the capability of StoryDistiller and large-scale dataset of apps, we are able to build a large and multi-dimension dataset with different kinds of data to enable different follow-up research directions.

Specifically, in this extension, we evaluate StoryDistiller on 150 apps (75 open-source and 75 closed-source apps) from the following two aspects: effectiveness evaluation of each phase of the proposed method and usefulness evaluation of the visualization outputs as a web service. The experimental results show that (1) for activity transitions, StoryDistiller outperforms the existing static methods such as IC3, Gator, and StoryDroid (7.8, 10.0, and 18.2 vs. **23.3** transition pairs on average); For activity coverage, StoryDistiller also performs the best compared with the above three static methods (38.7%, 33.7%, and 69.6% vs. **77.5%** on average) and the dynamic method (i.e., Stoat) (36.3% vs. **77.5%**). (2) StoryDistiller achieves around **80%** launch ratio of activities for each app on average on the 150 selected apps, while StoryDroid only launches about 55% activities due to the limitations of the pure static rendering method. Moreover, our rendered UI pages clearly show the actual functionalities of the activities compared with the ones that are obtained by manual exploration and achieve over 95% UI similarity. In addition, the user study shows that with the help of StoryDistiller, activity coverage has a significant improvement compared with exploration without StoryDistiller when exploring and reviewing apps.

In summary, we make the following main contributions:
- This research work aims to automatically generate the storyboards of Android apps. It assists app development teams including PMs, designers, developers, and app testers to quickly have a clear overview of other similar apps and target different tasks such as app exploration and app review.
- We leverage a hybrid approach to extract a comprehensive ATG for Android apps, and render UI pages dynamically with high UI similarity compared with the real ones.
- We propose a novel method to render UI pages by obtaining the required ICC data for launching each activity, minimizing unexpected errors when rendering UI pages (Algorithm 2 in § 4.3).
- Our comprehensive experiments demonstrate not only the effectiveness of the generated storyboards, but also the usefulness of our StoryDistiller with the extracted rich features for assisting app review and analysis.
- To enhance the usability of StoryDistiller, we visualize the storyboards with all rich features through a web service (Fig. 4). We also construct a multi-dimension dataset with different kinds of features based on StoryDistiller and enable several follow-up research directions, such as extracting commonalities across apps, recommending UI design and code, and guiding app testing. We will gradually release these datasets to enable different research applications [28].
- We released the code of StoryDistiller on GitHub for the community to facilitate the following works: https://github.com/tjusenchen/storydistiller

4. For the pure static method in StoryDroid, rendering the page is based on the static layout files or the transferred layout files for the dynamic/hybrid layouts, and no more other parameters are needed like ICC data using in StoryDistiller.

2 Motivating Scenario

We detail the typical app review process [29], [30], [9], [10], [12] with our StoryDistiller for Android apps in terms 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, and Twitter) based on the input keywords (e.g., social and chat) from Google Play Store. She then inputs all of the URLs of these apps into StoryDistiller which automatically download all of these apps with Google Play API [31]. StoryDistiller 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 StoryDistiller, 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 [32], [33].

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 StoryDistiller. By clicking the UI screen of each activity in the storyboard, StoryDistiller 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 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 StoryDistiller, he can easily locate the logic code.

Mallory is an Android tester who has to test the corresponding app based on Bob's implementation. By exploring StoryDistiller, she can understand the main logic and functionalities to generate test cases. For apps with multiple versions, StoryDistiller is able to identify the UI

components that have been modified between different app versions. Therefore, she can also reuse most of the test cases since different versions of a single app have many common functionalities. Reusing test cases is useful to improve the efficiency of app testing.

3 PRELIMINARIES

In this section, we briefly introduce the concept of Android Activity and Fragment, and the mechanism of inter-component communication (ICC).

3.1 Android Activity and Fragment

There are 4 types of components in Android apps (i.e., Activity, Service, Broadcast, and Receiver). Activity [19] and Fragment [26] render the user interface and are the visible parts of 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. Service is another important component of Android that is used to perform operations on the background such as playing music and handling network transactions. It does not has any UI.

3.2 Inter-component Communication

When an app intends to make inter-component communication (ICC), e.g., start a new activity B or connect to other apps from the current activity A, it requires to create an "Intent" object describing the task. If there is other data/messages required to be transferred from activity A to activity B, the parameters, such as action, category, and extra parameters can be stored in the "Intent" object or in the "Bundle" class, and transferred to activity B for successful launching. When activity B receives it, it can parse the data inside and use them to render the UI screen or conduct other transactions. If activity B does not receive the necessary data or the proper form of the necessary data, it could not be rendered successfully, sometimes even causing an app crash, usually "NullPointerException". Note that, one activity also can be started by other activities via Fragments and inner classes [27].

As shown in Table 1, the ICC data transferred between components are classified into two categories: *primitive attribute* and *extra parameter*. The primitive attributes are usually stated in the intent-filter element of an activity in the AndroidManifest.xml file, indicating that only the intents with specific attributes can launch the activity, such as actions to be performed (e.g., android.intent.action.VIEW), URI data to be operated on (e.g., vnd.android.cursor.dir/vnd.google.note), and special flags associated with the "Intent", etc. Primitive attributes can also be declared in the Java files. The extra parameters are usually declared in the Java files in an Intent object or Bundle class, indicating the data transferring to the target activity, which is also the necessary data to launch the target activity, in the form of <key, $type$> pairs where key is a String indicating the parameter name and $type$ indicates the data type of the value. For example, if an activity requires a specific "pid" (e.g., pid = 2) to be successfully launched, then the key refers to the parameter name "pid", and the $type$ refers to data type of 2 (i.e., Integer).

TABLE 1: Data types transferred in ICC

Category	SubCategory	Data Type/Description
Primitive Attributes	Action	String
	Category	Set<String>
	Data	String
	Type	String
Extra Parameters	Basic	<$key, type$>pair, where key refers to the parameter name, and $type$ indicates the data type of the value (e.g., Integer, String).
	Bundle	Set of <$key, type$>pairs, each of which is a basic extra parameter.

4 OUR HYBRID APPROACH (STORYDISTILLER)

StoryDistiller takes an *apk* as input and outputs the visualized storyboard (S) with rich features for the app. Fig. 2 shows the overview of our hybrid approach (named StoryDistiller): (1) First of all, StoryDistiller instruments the *apk* so that activities can be launched by third-parties. (2) *Static extraction* includes *ATG extraction*, which leverages static program analysis to obtain relatively complete ATG. Meanwhile, the required ICC data (i.e., Activity launching parameters shown in Table 1) can be extracted through control- and data-flow analysis (refer to Section 4.2.2). (3) *Dynamic UI page rendering* launches the activities registered in the app one by one with the extracted ICC parameters. Meanwhile, it can also augment ATG through *dynamic UI component exploration*. After that, we can obtain a comprehensive ATG with rendered UI pages. (4) Moreover, the other rich features, such as layout code, Activity code, UI component, and call graphs are collected. (5) StoryDistiller then visualizes the storyboard of the app with all the extracted features in a webpage.

4.1 APK Instrumentation

In terms of the *APK instrumentation*, we first decompile the target apk and set "exported=true" in the AndroidManifest.xml file for each activity to enable the launching process by third-parties. We then repackage it to a new installable APK file and sign it to ensure its usability. Note that the repackaged apps are only used for the experimental purpose, and all the experiments are conducted in a controlled environment. The repackaged apps will not be released for commercial use.

4.2 Static Extraction

Static extraction mainly contains two steps: ATG extraction and ICC data extraction for dynamic UI page rendering in the next phase.

4.2.1 ATG extraction

Activity transition in fragment and inner class are representative and widely-used components in real-world apps. According to our study on 150 randomly selected real apps

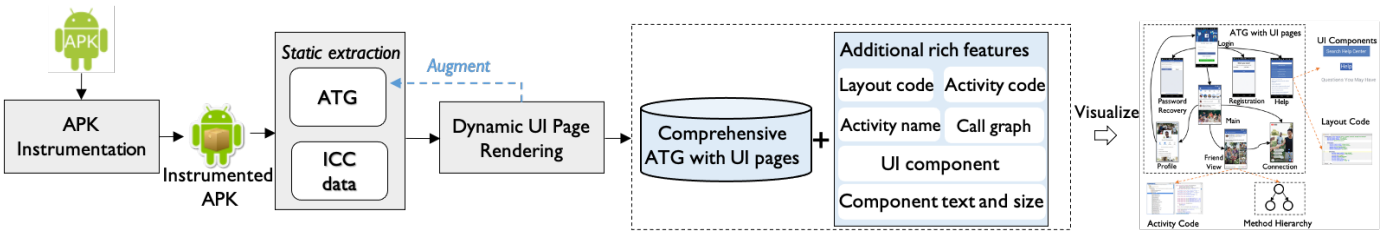


Fig. 2: Overview of StoryDistiller

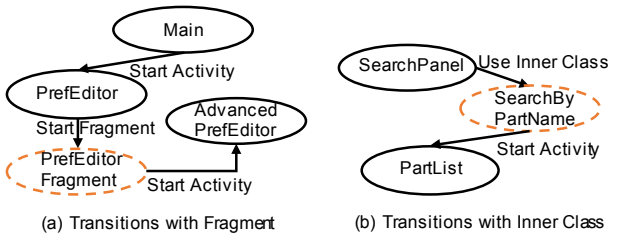


Fig. 3: Activity transitions between activities and fragment, inner class

(75 open-source and 75 closed-source apps used in RQ1 (§ 5.1)), we find 44 apps use Activity transitions in fragment and 84 apps use Activity transitions in inner class. Before extracting activity transitions in inner classes and fragments, we illustrate the transitions in them. Fig. 3 (a) is the sub ATG of *Vespucci* [34], a map editor. Firstly, activity Main starts PrefEditor, in which PrefEditorFragment is started. And PrefEditorFragment further starts AdvancedPrefEditor. Specifically, as shown in Listing 1, fragments can be added to an activity in two ways: (1) by invoking fragment modification API calls, e.g., "replace()", "add()", and further leveraging "FragmentTransaction.commit()" (lines 3-4) to start the fragment; (2) By using "setAdapter" (line 7) to display the fragment in a certain view (e.g., ViewPager). The started PrefEditorFragment then starts a new activity (i.e., AdvancedPrefEditor). Fig. 3 (b) shows the sub ATG of ADSdroid, where SearchPanel uses an inner class SearchByPartName to handle time-consuming operations as shown in Listing 2. After finishing the task, it starts an activity PartList by invoking "StartActivity()" (line 4). In this example, our goal is to extract activity transitions: Main→PreEditor, PreEditor→AdvancedPrefEditor, and SearchPanel→PartList.

Algorithm 1 details the extraction of ATG. Specifically, it takes as input an apk, and outputs the activity transition graph (atg). 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. 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* via getTargetAct() (lines 4-8). Specifically, for each explicit activity transition, the target activity is explicitly indicated in the Intent object where an intent variable usually either explicitly declares the callee activity or uses a variable defined before or other types of implementation to indicate the target activity. We first analyze which intent constructor it creates (Intent has various constructors to receive different kinds/numbers of parameters), and then track the parameter that indicates the target activity by data-flow analysis. Finally, we can obtain the target activity ($callee\_act$). 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 9-11). Take Fig. 3 (b) as an example, we add an edge SearchPanel→PartList to atg.

```
1 public class PrefEditor{... //Using replace/add
2   PrefEditorFragment pref = new PrefEditorFragment();
3   FragmentTransaction.replace(R.id.content,pref);
4   FragmentTransaction.commit();
5 }
6 public class PrefEditor{... // Using setAdapter
7   ViewPager.setAdapter(getSupFragmentManager(), new
      PrefEditorFragment());
8 }
```

Listing 1: Simplified code snippet of Fragment

```
1 public class SearchPanel{...
2   private class SearchByPartName extends Asynctack
      <>{...
3     Intent intent = new Intent(MainActivity.this,
      PartList.class);
4     startActivity(intent);
5   }
6 }
```

Listing 2: Simplified code snippet of Inner Class

If m is in a fragment, we construct the relation between the fragment ($caller\_frag$) and the target component (lines 12-13). Specifically, for each activity transition, we first locate the class c that starts a new activity according to m, and then check the super class of it. If it extends a fragment, we then set $caller\_frag = c$. In fact, in terms of extracting the target activities from explicit transitions, there is no difference between extracting activities started by activity and fragment. Note that this relation between the caller fragment and the target activity does not represent the actual component transition, we optimize it by identifying the activities that start the fragment in lines 18-21. Specifically, to identify the activities that bind a specific fragment, we investigate different types of methods that bind activities and the corresponding fragments, where fragments are operated (e.g., removed, added, replaced, and setAdapter) using specific APIs, and we can track specific APIs to identify the activity corresponding to a specific fragment. After that we update atg by merging fragment relations to construct the actual activity transitions (line 22). For example, as shown in Fig. 3 (a), we first obtain the relations PrefEditorFragment→AdvancedPrefEditor, PrefEditor→PrefEditorFragment, then we merge it to PrefEditor→AdvancedPrefEditor 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

Algorithm 1: Static ATG Extraction

Input: apk
Output: atg: Activity transition graph, including Activity and Service

```
1  atg ← ∅
2  cg ← getCallGraph(apk)
3  all_classes ← getAllClasses(apk)
4  foreach c ∈ all_classes do
5      methods ← getClassMethods(c)
6      foreach m ∈ methods do
7          if hasActivityTransition(m) then
8              callee_act ← getTargetAct(m)
9              if isInnerClass(c) then
10                 caller_act ← outerClass(c)
11                 atg.addPair(caller_act, callee_act)
12             else if isInFragment(m) then
13                 atg.addPair(caller_frag, callee_act)
14             else
15                 caller_acts ← getCallerAct(m, cg)
16                 foreach act ∈ caller_acts do
17                     act.addPair(act, callee_act)
           // Optimize atg
18         if startFragment(m) then
19             caller_acts ← getCallerAct(caller_frag)
20             foreach act ∈ caller_acts do
21                 atg.addPair(act, callee_frag)
22             updateATGIfNeeded(atg)
23 return atg
```

m to obtain all the activities that start the target activity ($callee\_act$), then add them to atg (lines 14-17).

4.2.2 ICC data extraction

As aforementioned in § 3.2, to successfully launch an activity, data that are required to render the target UI page should be provided, including the *primitive attributes* and *extra parameters* listed in Table 1. Algorithm 2 details the extraction process of ICC data. We highlight that the data-flow analysis for ICC data extraction is one of the core phases in StoryDistiller, which obviously improves the ability of UI page rendering (c.f. § 4.3).

As shown in Algorithm 2, it takes an *apk* as input, and outputs the ICC data required to launch each activity. Specifically, we first obtain the call graph, all class instances, and the AndroidManifest.xml file by decompiling the apk, and the output icc_data is initialized as an empty set (Lines 1-4). We then traverse the classes to identify activities. For each activity act, we use the function `getParamters()` to obtain the required parameters (including *primitive attributes* and *extra parameters*) for launching the activity (Lines 9-31). As for the **primitive attributes**, we obtain them (if any) from the manifest file by parsing the corresponding fields, such as "action" and "category", and then save it in $para$ (Lines 11-12). Sometimes primitive attributes are also declared in the source code, and the extraction method is similar to that of extra parameters.

Algorithm 2: ICC Data Extraction

Input: apk
Output: icc_data <act, $para$>: ICC data of each activity for Activity launching

```
1  icc_data ← ∅
2  cg ← getCallGraph(apk)
3  all_classes ← getAllClasses(apk)
4  mani ← getManifest(apk)
5  foreach c ∈ all_classes do
6      if isActivity(c) then
           // Get parameters for activity c
7          para ← getParameters(c, cg, mani)
8          icc_data = icc_data ⋃ <c, para>
9  Function getParameters(act, cg, mani):
10     para ← ∅
       // Get primitive attributes from manifest
11     attr, value ← getPrimitiveAttr(c, mani)
12     para ← para ⋃ <attr, value>
       // Get extra parameters from source code
13     methods_lc ← getLifecycleCallbacks(act)
14     foreach m ∈ methods_lc do
15         type, key ← null
16         para ← getExtras(m, para);
17     return para
18 Function getExtras(m, para):
19     if hasExtraParameters(m) then
20         extras ← getAllExtras(m)
21         foreach e ∈ extras do
22             key ← getKey(e)
23             type ← getValueType(e)
24             para ← para ⋃ <key, type>
25     else
26         m_callee ← getCalleeMethod(m, cg)
27         while m_callee ≠ null do
28             para ← getExtras(m_callee, para)
29             m_callee ← getCalleeMethod(m_callee, cg)
30     return para
31 return icc_data
```

As for the **extra parameter** extraction, we first identify methods related to activity lifecycle (denoted by $methods_{lc}$), such as *onCreate*() and *onStart*() since extra parameters in these methods are related to page rendering. For each lifecycle callback (i.e., method) m, if it invokes specific APIs (e.g., *getStringExtra*, *getBundle*) to get the ICC extra data from the previous activity, we obtain the key through backward data-flow analysis and the value type of each extra parameter based on the corresponding APIs. We then save them in $para$ (Lines 19-24). Specifically, as for the *key*, whose main purpose is to get the attached data transferred from the source activity to the target activity, therefore, it is usually presented using constant strings and can be directly extracted from the code according to the specific APIs. As for the *value type*, we can get it according to the specific APIs of value types such as getStringExtra and getBooleanExtra. For example, btd =getIntent().getStringExtra("returnKey1"), the

key we obtained is "returnKey1", the value type is "String", and the ICC data is saved as <returnKey1, String>, we will provide a string value for the key returnKey1 at runtime to launch the target activity. In some cases, the extra parameters are not directly declared in the lifecycle methods, but in the methods that the lifecycle methods invoke, which would also affect the UI rendering if not provided with proper parameters. To tackle this situation, we first obtain the methods that the lifecycle callbacks invoke according to the call graph cg (Line 26), and iteratively explore each method to obtain the potential extra parameters with their key and value types by invoking $getExtras$ method (Lines 28-29). After obtaining the parameters for activity act, we store it together its required parameters to icc_data for further UI page rendering and exploration.

4.3 Dynamic UI Page Rendering

Dynamic UI page rendering mainly contains two steps: *UI page rendering*, which launches each activity dynamically based on the extracted ICC data; *UI component exploration*, which augments static ATG by exploring all interactive components of each activity to identify more activity transitions together with UI pages. Note that the static UI page rendering method used in StoryDroid by leveraging layout code transformation may lead to a big visual difference between the rendered pages and the real pages like Fig. 9 (a), the reasons are explained in § 5.2.2. However, there are no such limitations in StoryDistiller which uses the dynamic UI page rendering with the ICC data extracted by the data-flow analysis.

4.3.1 UI page rendering

After generating the activity transitions between different pages, we now aim to render the corresponding UI pages by exploring each activity of the app. Our goal is to render/explore as many UI pages as possible to visualize the transitions between activities. To the best of our knowledge, basically, there exist two methods rendering/exploring the UI pages: (1) Dynamic app testing tools such as Monkey [22] and Stoat [23], which aim to explore as many UI pages as possible by dynamically running the apps to detect more bugs, however they are demonstrated to only achieve ~35% activity coverage, which is far away from representing the complete relations between activities. (2) Static UI page rendering. Chen et al. [1] proposed to render UI pages by first converting dynamic/hybrid layout to static layout since they found 62.3% apps construct their UI pages by adopting dynamic/hybrid layout, and developing a dummy app to launch each activity with the help of static layouts. However, the rendering largely relies on the layout conversion process, causing incomplete or error rendering of the UI pages if the conversion process is incomplete.

To this end, we propose to render and launch activities dynamically with the help of the extracted ICC data and Android toolkit, and take screenshots accordingly. This approach has several advantages over the existing methods: (1) It does not need to generate test cases to run the app like dynamic app testing, but directly launch all activities one by one, which addresses the limitation that the test cases may not reach all the activities successfully; (2) It considers the data transferred from the previous activity that is essential for rendering the current activity, which alleviates the limitation of improper conversion process in StoryDroid [1]. The detail of the rendering process is as follows.

For each activity, if it requires parameters to launch, we provide it with a random dummy value according to its required data types (e.g., String/Integer/Boolean). As for the dummy value, we extracted the data defined the layout files when exploring apps and randomly choose values from them for different data types. In this way, we can append all parameters needed for activity launching. For example, if the extracted parameters of one Activity are <"userid", Integer> and <"username", String>. We will use the command: "`adb shell am start -n pkg/pkg.activityname -ei userid 2 -es username Alice`" to launch the current Activity, where `-ei` and `-es` refer to the data types of the parameters are Integer and String, respectively. More required extra parameters can be appended. For other data types, there are also corresponding commands, such as `-ez` for Boolean and `-ef` for float. Besides, to eliminate side-effect between different activities during launching, we provide a fresh state for each activity by forcing stop the previous launched ones. For activities that fail to launch due to app crashes or permissions required, we dump the layout hierarchy of the current activity and analyze it to check whether it contains keywords (e.g., "has stopped" and "keeps stopping" for app crashes, "ALLOW" and "DENY" for permission requests). When the app crashes, we stop the app and set it to the original state (i.e., a fresh state for another activity to launch). When the app requests permission from users, we automatically grant it to make it render the UI page normally.

Note that, the activity that is actually launched may be different to what is intended to be launched. For example, we intend to launch an activity called "NewsDetailActivity", however this activity requires user credentials (e.g., user name and password). Without valid user credentials, it would jump to the "sign in" or "sign up" page. Thus the actual launched activity would be the "signInActivity". Considering such situations, to avoid assigning incorrect activity names to the launched UI pages, we obtain the current launched activity by retrieving the top activity from the back stack through the Android running system. This strategy also addresses the code obfuscation problem on activity names, which is better than the solution proposed in StoryDroid [1], i.e., inferring semantic names based on the layout tree similarity.

4.3.2 UI component exploration

Although the completeness of ATG is much better than the existing static method such as IC3 and dynamic method such as Stoat according to the comparison experiments in StoryDroid [1], some of the important activity transitions are still missing due to the limitation of the pure static method. In this paper, we propose to explore interactive components on each page and augment ATG. Specifically, when the UI page is rendered successful (§ 4.3.1), StoryDistiller follows two steps to conduct dynamic UI component exploration.

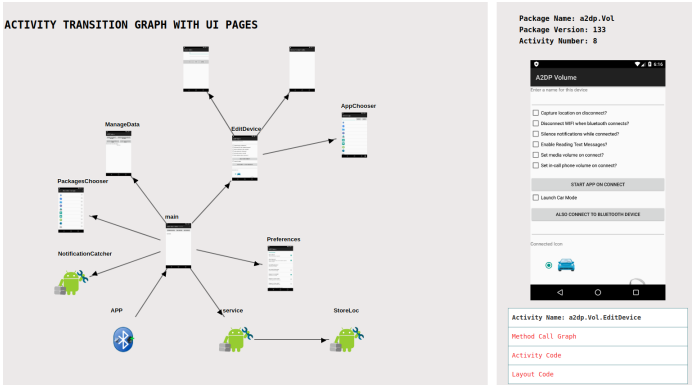


Fig. 4: Web service of StoryDistiller

Firstly, we parse the layout code of each rendered activity and extract each *interactive component* (e.g., *ImageButton*, *Button*, and *clickable TextView*) together with its attributes, including UI component id, component description, etc. Secondly, we trigger each interactive component on the rendered activity by using UIAutomator[35]. If the behavior triggers the launching of another activity and the transition is not included in the current ATG, we add the new explored transition pair into the ATG. By leveraging the hybrid ATG construction approach, we are able to obtain a more complete ATG for the demonstration of storyboards.

4.4 Rich Feature Extraction and Implementation

4.4.1 Feature Extraction

To visualize the storyboards of Android apps with all rich features, we highlight the extracted rich features for different software engineering tasks. Specifically, as shown in Fig. 2, we extract 8 kinds of features, including ATG, UI page, activity name, layout code, activity code, call graph, and UI components with their attributes. Among them, ATG, UI page, activity name, and call graph are extracted in § 4.2-§ 4.3 to achieve specific tasks. For activity code, we extracted the corresponding code by decompiling the APK file using the reverse-engineering tool. For layout code, we obtain them when rendering UI pages by dumping the layout for the current activity, which is the actual layout for the launched activities. For UI components and their attributes, we first identify the boundary and the attributes of each component (e.g., "Button" and "EditText") from the layout code, and crop each component according to the boundary.

4.4.2 Implementation

We implement StoryDistiller as an automated tool, which is written in 4K lines of Java code, and 3K lines of Python code. StoryDistiller is built on top of several off-the-shelf tools: IC3, jadx [36] and Soot [37]. We extend the Soot framework to extract inputs of UI page rendering, such as ATG and ICC data, 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 obtain the source code for Android apps. ApkTool (v2.4.1) [16] is used to repackaged the apk to implement the instrumentation. We dump the actual activity names for each UI page from the console through activity back stack [38]. For the few cases where activity names lack semantics and users have demand to obtain the inferred activity name, the method proposed in StoryDroid [39] can also be applied. The used Android emulator (Nexus 5X) is running on Genymotion (v3.0.0) with Android 8.0, 4G RAM, and 1920∗1080 resolution ratio. We use data-driven document (D3) [40] to visualize StoryDistiller's results, which provides a visualized technique based on data in HTML, JavaScript, and CSS. As shown in Fig. 4, the visualization [1] 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; (4) The components of each UI page with corresponding attributes, such as label and size; (5) The method call relations within each activity.

5 Evaluation of StoryDistiller

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

RQ1: Can StoryDistiller extract a more complete ATG in terms of more transitions and higher activity coverage compared with existing ATG exploration tools (i.e., IC3 [20], Gator [21], Stoat [23], and StoryDroid [1])?

RQ2: Can StoryDistiller render more UI pages with higher UI similarity compared with StoryDroid?

RQ3: Can StoryDistiller help explore and review the functionalities of Android apps effectively and efficiently?

5.1 RQ1: Effectiveness of Hybrid ATG Extraction

5.1.1 Setup

To investigate the capability of constructing ATG, we randomly download 75 apps from Google Play Store (closed-source apps) and 75 apps from F-Droid [41] (open-source apps) as subjects to demonstrate the effectiveness of ATG extraction on real-world apps. We compare StoryDistiller with four existing ATG exploration tools including three static methods (i.e., IC3 [42], Gator [21], and StoryDroid [1]), and one dynamic method, i.e., Stoat [23] which has been demonstrated to be more effective on app exploration than other tools such as Monkey [22]. For some closed-source apps, IC3 and Gator take more than one hour to extract ATG probably due to some internal errors, therefore, we set a timeout of 30 minutes for each app which is sufficient to explore most of the apps. For Stoat, we run each app for 30 minutes. As for the evaluation metrics, we use the number of *activity transition pair* and *activity coverage* to demonstrate the performance of each tool. "activity coverage" is computed as the number of unique activities in the ATG over the total number of activities declared in the app.

5.1.2 Results of RQ1

Activity transition pairs. Fig. 5 shows the result of tool ability in terms of extracting activity transition pairs. It can be seen that StoryDistiller outperforms the other three static tools for both open-source apps and closed-source apps. More specifically, StoryDistiller is able to extract 15.2 and 31.4 transition pairs on average for each open-source app and each closed-source app, receptively. Compared

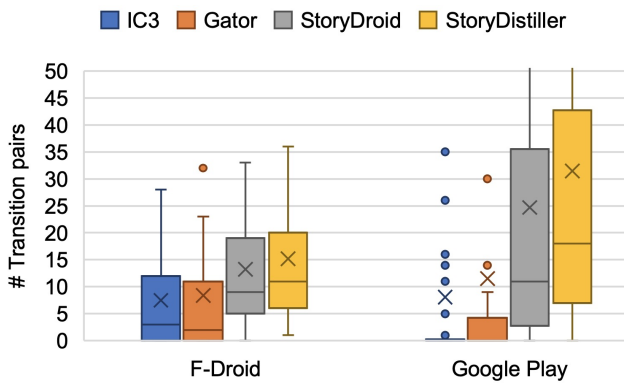


Fig. 5: Comparison of transition pairs

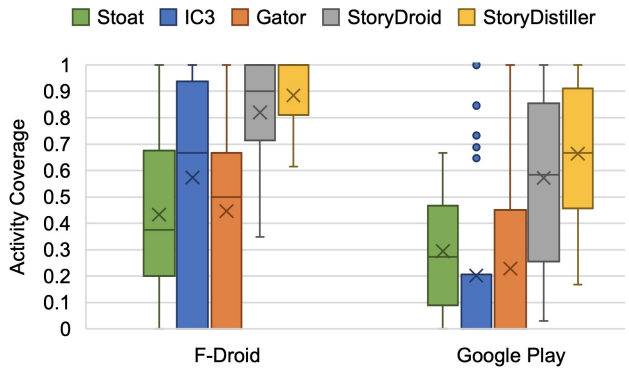


Fig. 6: Comparison of activity coverage

with StoryDroid, StoryDistiller improves over 20% transition pairs, which benefits from the proposed dynamic UI component exploration. Compared with IC3 and Gator, StoryDistiller increases more than twofold (7.78 for IC3, 9.96 for Gator vs. **23.30**) on all these selected apps.

As for StoryDistiller, it can extract activity transitions with respect to all the features such as fragments, inner classes, and callbacks. Since we extract transitions by using particular APIs (e.g., StartActivity, StartActivityForResult, and StartActivityIfNeeded) that start new activities by leveraging data-flow analysis, the extracted transitions are more accurate. To investigate the contribution of fragment and inner class to ATG, we record the number of apps that use fragment or inner class to start new activities. The result shows 44 apps use fragments and 84 apps use inner class to start new activities, indicating the popularity of using these two types to build activity transitions in real scenarios. Besides, the dynamic UI component exploration can also augment ATG. Even though, StoryDistiller sometimes may still miss some transitions due to the limitation of the underlying tools such as decompilation failures or extraction errors of certain classes. Besides, developers may self-define some methods to start new activities instead of using the default patterns (*c.geo* [43] open-source app), causing some activity transition pairs cannot be identified and extracted. Similarly, intent overloading [44] would also lead to missing activity transitions (*FBReader: Favorite Book Reader* [45]). To some extend, dynamic UI component alleviates this problem and augment the transitions effectively. Overall, the results shown in Fig. 5 demonstrate the effectiveness of StoryDistiller on extracting activity transitions over other existing tools.

Compared with IC3, StoryDistiller has advantages on inner classes, fragments, and callbacks when extracting activity transitions, which has been evaluated in StoryDroid [1]. However, according to our investigation, for intent overloading with complex parameters, IC3 can extract partial activity transitions statically. Therefore, to obtain a comparatively complete ATG and maximize the activity coverage, we implemented StoryDistiller by integrating the transition results of IC3.

Activity coverage. Fig. 6 depicts the activity coverage results of each tool. Compared with the dynamic method, on average, StoryDistiller outperforms Stoat in terms of activity coverage, achieving 88.5% (vs. 43.2%) and 66.4% (vs. 29.4%) coverage on open-source apps and closed-source apps, respectively. In addition, StoryDistiller costs much less time (i.e., 8.50 minutes on average) to extract and render the activities than Stoat (i.e., 30 minutes). The time cost includes the apk instrumentation and UI page rendering. As for the comparison results with static methods, the performance trend is similar to that of activity transition pairs. StoryDistiller still outperforms other tools, achieving nearly 80% coverage on average. Compared with StoryDroid, StoryDistiller improves over 10% activity coverage.

StoryDistiller 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 apks, causing failures in extraction of activity transition and coverage. That situation is more severe in closed source apps due to packing [46] and code obfuscation techniques [47], [48]. (2) Another reason is the dead activities (no transitions), such as unused legacy code and testing code in apps. We also investigate the reasons why dynamic exploration tools such as Stoat achieve 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 in low coverage.

Answer to RQ1. StoryDistiller outperforms the static methods (e.g., IC3, Gator, and StoryDroid) in terms of activity transition pairs (**23.3** vs. 7.8 in IC3, 10.0 in Gator, and 18.2 in StoryDroid), and the dynamic method (e.g., Stoat) in terms of activity coverage (**77.5%** vs. 36.3% in Stoat). Therefore, StoryDistiller is able to obtain a more complete activity transition graph compared with existing tools.

5.2 RQ2: Effectiveness of UI Page Rendering

5.2.1 Setup

To investigate the effectiveness of UI page rendering, we compare StoryDistiller (dynamic method) and StoryDroid (static method) in terms of the ratio of rendered pages and the UI similarity of rendered pages, by using the 150 Android apps in RQ1. Specifically, (1) we first investigate the

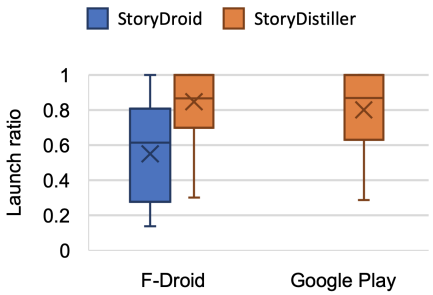


Fig. 7: Comparison the Launch ratio of activities between StoryDroid and StoryDistiller

ratio of UI pages (activities) that are successfully launched in each app, denoted by $LaunchR$.

$$LaunchR_i = \frac{N_i^{Launched\_act}}{N_i^{All\_act}} \times 100\%$$

Where $N_i^{All\_act}$ indicates the number of activities declared in the AndroidManifest.xml file in the i^{th} app. (2) We further investigate whether the functionalities of the launched pages are clearly displayed, i.e., users can easily and clearly understand the functionality through the UI pages. To do it, we compute the visual similarity between the rendered UI pages and the real UI pages to demonstrate the rendering ability of StoryDistiller in practice. Note that the real UI pages are obtained by Monkey [22].

5.2.2 Results of RQ2

Launch ratio. Fig. 7 shows the distribution of the launch ratio of each app. Note that since StoryDroid fails to launch activities due to the apk compilation failures for most of the closed-source apps (only 6 out of 75 closed-source apps), to avoid introducing bias based on such a small dataset, we only show its result on open-source apps in the box plot. The reasons for such a low launch ratio on closed-source apps are also described in this section. We can see that on average, over 80% activities (i.e., 82.37% for open-source apps, 80.14% for closed-source apps shown in Table 2) can be launched successfully by StoryDistiller in our dataset, the remaining ones encounter crashes when being launched, usually caused by "NullPointerException" or "ClassNotFoundException". Besides, we further investigate the contribution of ICC data to activity launching, with the help of the extracted ICC data by StoryDistiller, there are 37.69% additional activities for closed-source apps (29.48% for open-source apps) being launched successfully.

While StoryDroid only launches 55% activities for open-source apps, as shown in Table 2. Fig. 7 also indicates that apps from Google Play are more likely to get lower launch ratio, i.e., more cases at the bottom. Almost all the launch ratio of open-source (i.e., 73 apps) are over 50%, and the lowest launch ratio is 30% in our dataset shown in Table 2. However, as for closed-source apps, there are some cases (i.e., 9 apps) whose launch ratio are below 50%. Specifically, the lowest rate achieves 28.27% launch ratio shown in Table 2. A possible reason may be the more complex functionalities in closed-source apps, which is evidenced by the number of transition pairs in Fig. 7 (a).

TABLE 2: Comparison the Launch ratio of activities including average, minimum, and maximum ratio between StoryDroid and StoryDistiller

Method		Static (StoryDroid)	Dynamic (StoryDistiller)	
Sources		F-Droid	F-Droid	Google Play
Launch Ratio	Avg.	55%	82.37%	80.14%
	Min.	13.64%	30%	28.27%
	Max.	100%	100%	100%

The main reason that StoryDroid fails to launch activities for most of the closed-source apps is apk compilation failures listed below, which are all mitigated by StoryDistiller. (1) *Due to missing necessary configuration files*. StoryDroid supports rendering UI pages for open-source apps because it requires to obtain the configuration file of the project (e.g., *build.gradle*[5]) which includes necessary library dependencies and other configurations, however, the configuration file only appears in the source code. It cannot be obtained even by decompiling the apk files. (2) *Due to user-defined components [49], complex grammar representations, and resource file errors (e.g., XML layout files), etc.* Even for open-source apps, it is still difficult for StoryDroid to obtain user-defined components and complex grammar representations (e.g., Syntactic Sugar [50]) by using the proposed static method, causing rendering failures in these UIs. Last but not least, errors caused by resource files sometimes occur when we build the dummy app. These limitations cause StoryDroid ineffective in many apps in our dataset.

Visual similarity. We compare the visual similarity between the real pages and the rendered UI pages by StoryDistiller and StoryDroid to demonstrate the quality improvement of rendered UI pages based on the 150 apps in RQ1. We obtain real pages by leveraging Google Monkey [22] to dynamically explore UI pages and take screenshots, and select the overlapping activities of real ones and rendered ones by their activity names. We use two widely-used similarity metrics [51], [1], [52]: mean absolute error (MAE) and mean squared error (MSE) to measure the visual similarity.

The result shows that StoryDroid only achieves about 80% UI similarity, while StoryDistiller achieves 96.5% and 91.6% UI similarity in terms of MAE and MSE respectively. Fig. 8 shows some real examples rendered by StoryDistiller, we can see that StoryDistiller can render UI pages with various types of components, such as RadioButton and ListView. Even for the UI pages using complex design structure or theme, multi-components, self-defined components, multi-images, or rich page color, StoryDistiller still performs well in most cases. Compared with StoryDistiller, StoryDroid only uses testing data to replace real data for components such as ListView and GridView, which decreases the UI similarity compared with the real UIs. As shown in Fig. 9, StoryDroid cannot render such complex design structure or theme due to lack of data dependency, which would lose some main functionalities. For example, Fig. 9 (a), rendered by StoryDroid, shows that "No hosts created yet" without showing the main structure of the UI page due to lack

5. A *build.gradle* file will be generated when creating a new Android project through Android Studio. We take this file as the default *build.gradle* when closed-source apps render the UI pages in StoryDroid.

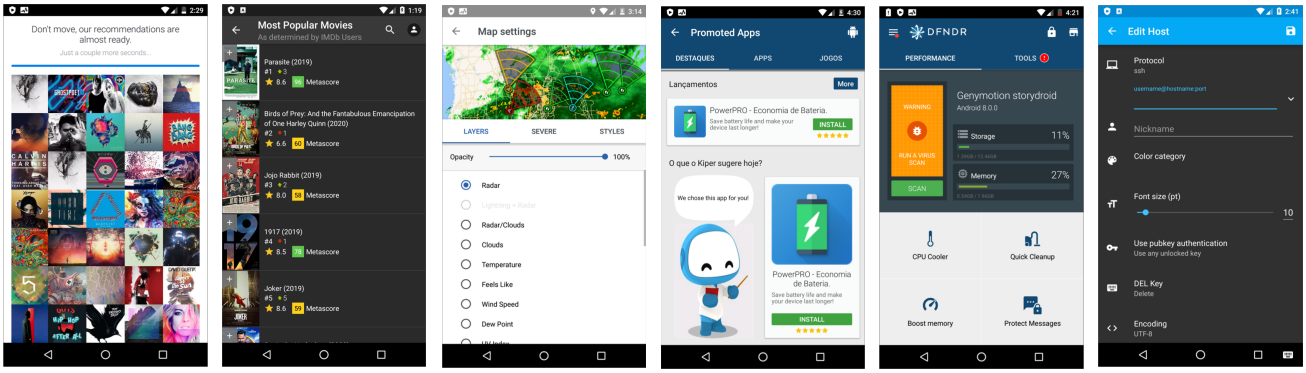
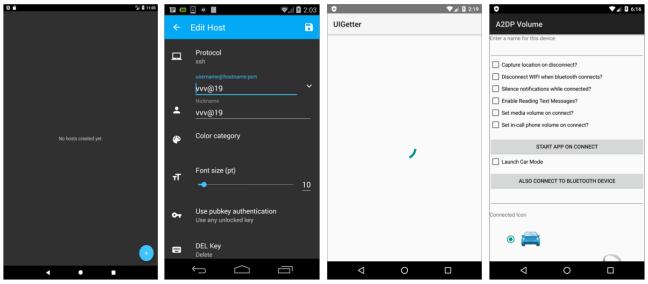
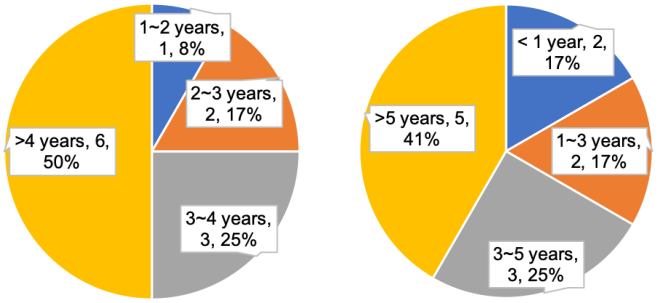


Fig. 8: Examples of successfully rendered UI pages with diverse components



(a) EditHost (StoryDroid) (b) EditHost (StoryDistiller) (c) EditDevice (StoryDroid) (d) EditDevice (StoryDistiller)

Fig. 9: Examples of the same UI pages rendered by StoryDroid and StoryDistiller



(a) Years of Android device usage (b) Years of conducting Android-related work

Fig. 10: Distribution of participants

of history data for the EditHostActivity. In contrast, Fig. 9 (b), rendered by StoryDistiller, displays all functionalities dynamically even for the save button (on the top right) with the real theme. Similarly, Fig. 9 (c) cannot be rendered like Fig. 9 (d) to demonstrate the real functionalities.

As we investigated, sometimes errors still occur in the rendered UI pages by StoryDistiller due to data loss in practice. We summarized six types as follows and show some real cases in Fig. 11. (1) *Remote server data*. Fig. 11 (a) shows an activity named "PaylistActivity", however, the detailed pay list information is lost since they are stored in the remote server. (2) *Local database data*. Fig. 11 (b) shows the profile of a user without detailed data since the data should be loaded from the local SQLite database. (3) *Unauthorized access to webpages*. As shown in Fig. 11 (c), it is a WebView page shows the terms and condition, however, due to unauthorized access, the WebView page fails to load. (4) *Hardware support.* Fig. 11 (d) is an app relying on hardware support (i.e., NFC). However, we conduct the UI page rendering on an Android emulator without required hardware support. (5) *Login authentication.* Fig. 11 (e) failed to be rendered due to the login authentication. Only users with valid authentication information can get access to the page, as indicated by the activity name. (6) *Long loading time.* Fig. 11 (f) is a map app. Due to the inadequate rendering time, the map is not rendered completely.

Although some specific data is not loaded or rendered successfully, the rendered information together with the activity names are still enough for users to understand the functionality of these pages. For instance, for the cases in Fig. 11 (a) (b) (f), we still can know the core logic of the activities. For the other three cases (i.e., Fig. 11 (c) (d) (e)), the activity names contain rich semantics, which can help users understand the core logic.

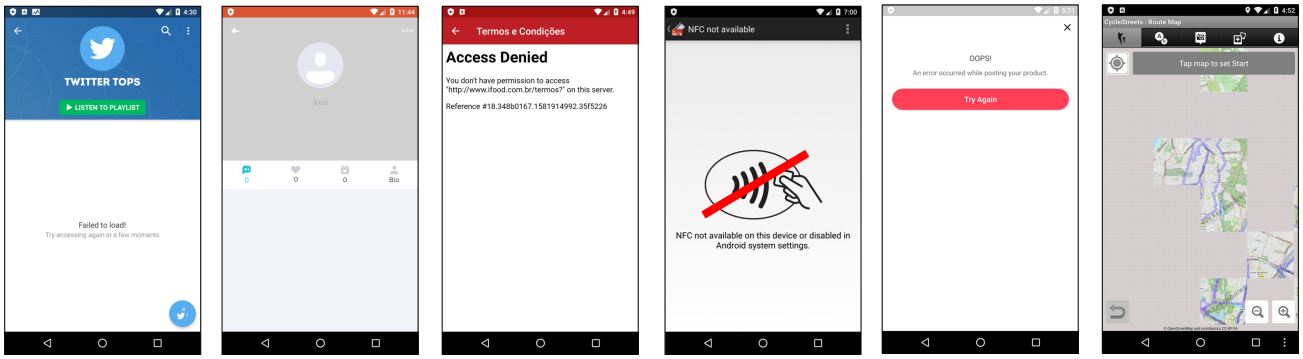
Answer to RQ2. StoryDistiller achieves ~80% launch ratio of activities for each app on average, which is much better than StoryDroid with only ~55% launch ratio when rendering UI pages on our dataset. Moreover, the rendered UI pages by StoryDistiller achieve a high UI similarity compared with StoryDroid (~80% vs ~95%).

5.3 Usefulness Evaluation of StoryDistiller

Apart from effectiveness evaluations, we further conduct a user study to demonstrate the usefulness of StoryDistiller. Our goals are to check whether StoryDistiller can help explore and review the functionalities of apps effectively and efficiently.

Dataset of user study. We randomly select 4 apps (i.e., Bitcoin, Bankdroid, ConnectBot, and 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 12 people including 2 professors, 2 postdocs, and 6 Ph.D students from our



(a) PaylistActivity (b) ProfileActivity (c) TermsAndCondition (d) MainActivity (e) ConfirmationAfterLogin (f) CycleStreets

Fig. 11: Examples of rendered pages with clear functionalities but with some data loss

TABLE 3: User study results of app exploration and review. $*$ denotes $p < 0.01$ and $**$ denotes $p < 0.05$.

Metrics	Manual Exploration	StoryDistiller
Avg. Time (min)	5.47	2.85*
Avg. Coverage	39.06%	88.30%*
Satisfactoriness (1-5)	3.99	4.48**

university and 2 industry staff from local companies 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. Detailed distribution is shown in Fig. 10. They never use these apps before. They come from different countries, 1 from USA, 4 from China, 4 from European countries (e.g., Spain, Germany), and 3 from 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 8.0). 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 and review the 4 apps separately to finish the tasks below. Note that for each category, each participant explored one app with StoryDistiller, and the other without StoryDistiller. To avoid potential bias, the order of app category, and the order of using StoryDistiller or not using are rotated based on the Latin Square [53]. This setup ensures that each app is explored by multiple participants with different development experience. We told each participant to complete the task with the given apps: 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) [54], and understand the app functionalities with StoryDistiller; After the exploration, participants were asked to rate their satisfactoriness in exploration (on the 5-point likert scale with 1 being least satisfied and 5 being most satisfied). All participants carried out experiments independently without any discussions with each other. After performing the task, they were required to write some comments about our tool.

Experiment results. As displayed in Table 3, the average activity coverage of manual exploration is quite low (i.e., 39.06%), showing the difficulty in exploring app functionalities thoroughly by manual exploration. However, the participants' satisfactoriness of completeness of exploration is high (i.e., 3.99 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, StoryDistiller achieves over 2 times more activity coverage (**88.30%** vs. 86.50% in StoryDroid) with less time cost (**2.85 minutes** on average vs. 2.5 minutes in StoryDroid) to help understand the app functionalities. According to the participants' feedback, the average satisfactoriness of StoryDistiller is **4.48** (vs. 4.40 in StoryDroid), which represents the usefulness of helping participants explore and understand app functionalities. To understand the significance of the differences between without and with StoryDistiller, we carry out the Mann-Whitney U test [55], which is designed for small samples. The result in Table 3 is significant with p-value < 0.01 or p-value < 0.05.

6 Dataset and Possible Applications

As aforementioned, StoryDistiller is a fundamental tool which constructs a multi-dimension dataset (e.g., app storyboards and UI components). Such a rich dataset can be used to expand the horizon of current mobile app research. In this section, we discuss several application scenarios by leveraging this dataset.

6.1 UI Design Recommendation and Layout Code Generation

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. For the tasks of UI design recommendation [56] and layout code generation [24], our dataset provides a large set of diverse UI pages, as well as the corresponding layout code. The diversity of the collected data depends on StoryDistiller's ability of thoroughly exploring apps' UI pages. Additionally, it is crucial to provide real UI pages for the UI design recommendation task. Based on the results of *ATG extraction* (§ 5.1) and *UI page rendering* (§ 5.2), StoryDistiller is able to obtain a high activity coverage compared with dynamic

testing tools and a high successful rate of UI page rendering. Moreover, the rendered UI pages are almost same as the real ones that users would observe.

The UI pages with attributes in our dataset can assist both UI designers and developers. Such a dataset bridges the gap across the abstract activities (text), UI pages (image) and detailed layout code (i.e., activity $\rightarrow$ UI page $\rightarrow$ layout code) so that they can be searched as a whole. Due to such mapping relation, UI/UX designers can directly use keywords (e.g., "Login" and "Search") to search for the UI images by matching the activity name of the UI in our dataset. The searched images can be used for inspiring their own UI design. The UI developers can also benefit from searching our dataset for UI implementation. For another application scenario, given the UI design image from designers, developers can search for the similar UIs in our dataset by computing the image similarity. As each UI page in our dataset is also associated with corresponding runtime UI code, developers can choose the most related UI page in the candidate list and then customize the UI code for their own purpose to implement the given UI design.

Additionally, by training a neural machine translator, we are able to translate a UI design image to a GUI skeleton. Chen et al. [24] collected the training data based on the dynamic testing tool, Stoat. However, according to the experimental results of ATG generation, we find that StoryDistiller covers 2 times more activities than Stoat with less time. Consequently, the results are limited to the diversity of the training data used in [24]. Our constructed dataset of UI pages are more comprehensive with diverse UI designs.

6.2 UI Component Recommendation

UI component sharing provides an opportunity to learn about GUI designs, gain design inspiration and understand design trend [57]. To enable the recommendation task of GUI components, our dataset collected a large number of separate UI components (e.g., "Button") together with their attributes and the corresponding bank-end design code. Based on them, we highlight some typical tasks or potential application scenarios as follows. (1) Alice aims to design a social media app and wants to decide the style of the buttons so that it can fit for the theme of such a social media app. With the constructed dataset, she can search for hundreds of buttons to get inspirations. According to the candidates returned by the dataset, she can choose the most attractive one as the final decision for her own apps. (2) Apart from the style of UI component, the size and color are also provided to Alice. Therefore, she may observe that social apps usually use larger size with bright color buttons for most social media apps. (3) Based on the results of multiple-time searching, Alice may also understand the design trend of UI components in one app category, which is also helpful for developing apps in specific categories.

6.3 Code Search

When developers implement their own apps, aiming to ensure the competitive edge in the markets, they usually attempt to get inspirations from the similar components (e.g., Activity) implemented in other apps, because the components with the same semantic name have a great probability to own similar logic and architectures (e.g., method hierarchy). To enable such a code search task, our constructed dataset also collects the logic code with Activity names. Firstly, we divide the apps based on their app categories, such as finance, social media, and news since the apps in the same category would contain more common features. Secondly, we store the activities if they have the same semantic activity name, such as LoginActivity, RegActivity, AboutActivity, and EditActivity.

For example, Bob is a junior app developer. For the login activity, he may only implement the basic logic, i.e., collect user' inputs and validate whether the inputs are consistent with the information stored in the server or the database. With the help of our constructed dataset, he can search for the similar implementation by the same Activity name, i.e., *LoginActivity* or just *Login*. After searching, he would note that he should also validate the format before collecting the users' inputs, which is a typical specification. In this case, the logic code with same name could help to improve the quality of their own apps and customize more interesting features.

6.4 StoryDistiller for App Testing

App GUI testing. According to many previous studies [58], [59], there are only about 40% activity coverage for most dynamic GUI app testing tools such as Monkey and Stoat, mainly due to lack of improper user input complex constraints. Thanks to the relatively complete ATG constructed by StoryDistiller, we can leverage it to explore more activities and enhance the exploration capability of transition-based dynamic testing tools. For example, when apps are under testing by using Monkey, we can differentiate the transitions that are never explored by Monkey by comparing the transitions and covered activities. For the uncovered transitions, based on our ATG, we can directly launch the target activities and make the testing tool start to explore from this new state (using their own exploration strategy) to explore more state and detect more bugs.

App regression testing. Reusing test cases is useful to improve the efficiency of regression testing for Android apps [60]. StoryDistiller can help guide app regression testing by identifying the ATG and UI components that have been modified. Note that, different versions of a single app have many common functionalities, which means most of the UI pages in the newer version are the same as the previous version. The ATGs of different versions can be easily used to demonstrate the common functionalities. Meanwhile, StoryDistiller stores the mapping relation between UI page and the corresponding layout code, therefore, analyzers can obtain the modified UI components by analyzing the differences of layout code, and further update the test cases accordingly. 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.

7 LIMITATIONS

In this section, we discuss the limitations of StoryDroid.

Incomplete features due to the underlying tools. 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.

Failures in UI page rendering. Although StoryDistiller achieves ~80% launch ratio of activities for each app on average, some UI pages still cannot be rendered successfully due to several errors. (1) Some activities require valid authentication information to launch, that is, they will check whether the current state owns valid authentication (e.g., successful login in) before rendering the page, if the activity tries to be launched without valid authentication, it may redirect to the sign-in or sign-up page. Such scenario is an open challenge in Android app testing, unless the testers provide the login information before hand to enable the login process, then the app can continue explore the pages that require valid authentication. Thus, StoryDistiller would fail to render such kind of activities. (2) Although we provide the required ICC data as the activity launching parameters, some activities still need to load other required data from local storage (e.g., SharedPreference, SQLite Database) or remote servers. StoryDistiller cannot provide this kind of required data so far, causing failures when launching this kind of activities.

Incomplete activity presentations due to fragments. As aforementioned in the paper, an activity may have multiple fragments in practice. First of all, it is possible to define in the static layout file of an activity that it contains fragments (i.e., static binding), and fragments are treated as views to render the activity. While developers can also choose to bind the fragments (e.g., add, delete, and replace) of an activity at runtime (i.e., dynamic binding). As for the current version of StoryDistiller, it only records one UI page per activity with static fragments. If the current activity uses the static binding method to bind fragments, StoryDistiller can leverage the proposed hybrid method to render the activity with fragments. However, if the fragments are integrated into the activity at runtime triggered by users or specific operations, StoryDistiller cannot record the changes for different fragments in one activity so far.

8 Related Work

Assist 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 [51] reverse-engineer the UI screenshots by image processing techniques. More powerful deep-learning based algorithms [24], [61], [62] are further proposed to leverage the existing big data of Android apps. Retrieval-based methods [63], [64] are also used to develop the user interfaces. Reiss [63] 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 [51], [24], [61], [62] is all static layout, which conflicts with our observation in Section 3 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 recommend UI design [57] and explore issues between UI design and its implementation. Moran et al [65] 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 [64] customizes Reiss's method [63] 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 like Stoat, leading to the loss of information.

Assist app comprehension by reverse engineering. The process of reverse engineering of Android apps is that researchers rely on the state-of-the-art tools (e.g., Apktool [16], Androguard [17], dex2jar [66], Soot [37]) 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 [67]. It also can be used to extract features for Android malware detection [68]. However, reverse engineering only has the basic functionality for code review. Different from the general reverse engineering with plain decompiled code, our work extract more abstract representations, i.e., storyboard of each app to give the overview of app functionalities and mappings between the UI page and the corresponding layout code. Such storyboard can directly help product manager and designers who are of no technical expertise to understand competitor apps.

Assist Android app analysis. Many static analysis techniques [15], [42], [20], [69], [70], [4], [5], [8], [6] have been proposed for Android apps. A^3E provides two strategies, targeted and depth-first exploration, for systematic testing of Android apps [15]. 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 [42], and it determines

most Intent attributes to component matching. ICC [20] significantly outperforms Epicc on the extraction ability of inter-component communication by utilizing the solver for MVC problems based on the proposed COAL language. FlowDroid [69] and IccTA [70] extract call graphs based on Soot for data-flow analysis for detecting data leakage and malicious behaviors [68], [71], [72], [73], [74], [75], [6], [8], [7]. Liu et al. [76] utilized program analysis to understand the patterns that cause functional and nonfunctional issues and proposed a static analysis tool to detect two most common patterns of wake lock misuses. Wei et al. [77] combined program analysis and NLP techniques to prioritize Lint warnings by leveraging app user reviews. Dong et al. [78] proposed time-travel testing for Android apps that can transit to the state it explored before when needed. Wei et al. [79] proposed an approach that automatically learns API-device correlations of compatibility issues induced by fragmentation from existing Android apps. Yan et al. [80] proposed multi-entry testing for Android apps by analyzing the constraints for launching an activity and the solved constraints are used to launch the activity through a third-party app. They did not focus on ATG construction, instead, they focused on the construction of Activity Launching Models (ALM) by a static method (i.e., starting with a coarse-grained ATG mentioned in their paper). By contrast, extracting a relatively comprehensive ATG is one of the most important goals in our work, and we not only statically extract the transitions between activities, fragments, and inner classes, but also dynamically augment ATG to construct a comprehensive graph. In terms of dynamic exploration, their goal is to adjust the weights of their Activity Launching Context (ALC) dynamically to explore apps and find bugs by leveraging their constructed Activity Launching Model. Instead, our goal is to augment the transition graph extracted by the pure static method in the previous work [1] by traversing the actionable components in the UI page to explore as many transitions as possible. As for the activity launching, their method required to build a dummy app to launch activities due to the limitation of launching via adb, while our work addresses this problem by instrumenting the app, thereby can launch the activity directly from the console via adb instead of a dummy app used in [80]. Compared with them, we provide another novel solution to assist Android app testing, i.e., reveal the relations between different components together with rich attributes to help understand the semantic and functionality of apps.

9 Conclusion

In this paper, we propose StoryDistiller, a system to distill visualized storyboards of Android apps with rich features by extracting relatively complete ATG and rendering UI pages dynamically with the help of the extracted ICC data. Such a storyboard benefits different roles (i.e., PMs, UI designers, developers, and testers) in the app development process and analysis. The extensive experiments and user study demonstrate the effectiveness and usefulness of StoryDistiller. Based on the outputs of StoryDistiller, we constructed different kinds of large-scale datasets to bridge the gap across app activities (descriptive text), UI pages (image), and implementation code (source code). In the future, we will further explore these potential applications, and also extend our approach to other platforms such as iOS apps and desktop software for more general usage.

Acknowledgments

We appreciate all the reviewers for their valuable comments. This work was partially supported by the National Natural Science Foundation of China (No. 62102284, 62102197).

References

[1] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for Android apps," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 596–607.

[2] (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/

[3] (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/

[4] 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*. ACM, 2018, pp. 408–419.

[5] 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*. ACM, 2018, pp. 485–496.

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

[7] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global Android banking apps," in *Proceedings of the 42nd International Conference on Software Engineering*. IEEE Press, 2020, pp. 596–607.

[8] 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.

[9] 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.

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

[11] (2015) Competitor analysis before launching a mobile app startup. [Online]. Available: https://growthbug.com/competitor-analysis-before-launching-a-mobile-app-startup-f2f6a19f21b7

[12] 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.

[13] (2018) How Much Does an App Cost. [Online]. Available: https://savvyapps.com/blog/how-much-does-app-cost-massive-review-pricing-budget-considerations

[14] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "A comparison of Android reverse engineering tools via program behaviors validation based on intermediate languages transformation," *IEEE Access*, vol. 6, pp. 12 382–12 394, 2018.

[15] 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.

[16] (2018) A tool for reverse engineering Android apk files. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[17] (2018) Reverse engineering of Android applications. [Online]. Available: https://github.com/androguard/androguard

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

[19] (2018) Android documentation: Activity. [Online]. Available: https://developer.android.com/reference/android/app/Activity

[20] 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.

[21] (2020) Gator: Program analysis toolkit for Android. [Online]. Available: http://web.cse.ohio-state.edu/presto/software/gator/

[22] (2018) Google Monkey for Testing. [Online]. Available: https://developer.android.com/studio/test/monkey

[23] 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.

[24] 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.

[25] 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.

[26] (2018) Android Fragment. [Online]. Available: https://developer.android.com/reference/android/app/Fragment

[27] (2018) Java Inner Class. [Online]. Available: https://www.tutorialspoint.com/java/java_innerclasses.htm

[28] (2020) Overview of StoryDistiller. [Online]. Available: https://sites.google.com/view/storydistiller/

[29] (2018) Mobile app development process. [Online]. Available: https://thebhwgroup.com/blog/mobile-app-development-process

[30] (2018) 4 steps to develop your app idea. [Online]. Available: http://apptology.com/blog/tag/mobile-app-storyboard/

[31] (2014) Crawl and download apps from Google Play. [Online]. Available: https://github.com/dflower/google-play-crawler

[32] (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

[33] (2018) Uninvited Redesigns. [Online]. Available: https://uninvitedredesigns.com/

[34] (2018) Vespucci. [Online]. Available: https://play.google.com/store/apps/details?id=de.blau.android

[35] A. U. Automator. (2020). [Online]. Available: https://developer.android.com/training/testing/ui-automator

[36] (2018) Dex to Java decompiler. [Online]. Available: https://github.com/skylot/jadx

[37] (2018) Soot: A Java optimization framework. [Online]. Available: https://github.com/Sable/soot

[38] (2020) Component stack. [Online]. Available: https://developer.android.com/guide/components/activities/tasks-and-back-stack?hl=en

[39] (2018) Overview of StoryDroid. [Online]. Available: https://sites.google.com/view/storydroid/

[40] (2018) D3.js. [Online]. Available: https://d3js.org/

[41] (2018) F-droid market. [Online]. Available: https://f-droid.org/en/packages/

[42] 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.

[43] (2020) c.geo. [Online]. Available: https://f-droid.org/wiki/page/cgeo.geocaching

[44] (2020) Java method overloading. [Online]. Available: https://www.w3schools.com/java/java_methods_overloading.asp

[45] (2020) FBReader: Favorite Book Reader. [Online]. Available: https://play.google.com/store/apps/details?id=org.geometerplus.zlibrary.ui.android

[46] (2018) Android Packer Tehchniques. [Online]. Available: http://www.ninoishere.com/android-packer/

[47] (2019) Proguard. [Online]. Available: https://www.guardsquare.com/en/products/proguard

[48] (2019) DashO. [Online]. Available: https://www.preemptive.com/products/dasho/overview

[49] U. defined Component. (2017). [Online]. Available: https://documentation.alphasoftware.com/pages/Guides/Mobile%20and%20Web%20Components/Custom/User-defined%20Components.xml

[50] J. Syntactic Sugar. (2020). [Online]. Available: https://en.wikipedia.org/wiki/Syntactic_sugar

[51] 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.

[52] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of Android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, 2019.

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

[54] 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.

[55] (2018) Mann-Whitney U test. [Online]. Available: http://www.statisticssolutions.com/mann-whitney-u-test/

[56] C. Bernal-Cárdenas, K. Moran, M. Tufano, Z. Liu, L. Nan, Z. Shi, and D. Poshyvanyk, "Guigle: a gui search engine for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 71–74.

[57] C. Chen, S. Feng, Z. Xing, L. Liu, S. Zhao, and J. Wang, "Gallery dc: Design search and knowledge discovery through auto-created gui component gallery," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–22, 2019.

[58] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[59] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

[66] (2018) Tools to work with Android .dex and Java .class files. [Online]. Available: https://github.com/pxb1988/dex2jar

[67] (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

[68] 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.

[69] 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.

[70] 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.

[71] 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.
[72] 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.
[73] 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.
[74] 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.
[75] 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.
[76] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 396–409.
[77] L. Wei, Y. Liu, and S.-C. Cheung, "Oasis: prioritizing static analysis warnings for android apps based on app user reviews," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 672–682.
[78] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of android apps," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering, ICSE*. ACM, 2020.
[79] L. Wei, Y. Liu, and S.-C. Cheung, "Pivot: learning api-device correlations to facilitate android compatibility issue detection," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 878–888.
[80] J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of android applications by constructing activity launching contexts," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering, ICSE*. ACM, 2020.

Sen Chen (Member, IEEE) is an Associate Professor in the College of Intelligence and Computing, Tianjin University, China. Before that, he was a Research Assistant Professor in the School of Computer Science and Engineering, Nanyang Technological University, Singapore. Previously, he was a Research Assistant of NTU from 2016 to 2019 and a Research Fellow from 2019-2020. He received his Ph.D. degree in Computer Science from School of Computer Science and Software Engineering, East China Normal University, China, in June 2019. His research focuses on Security and Software Engineering. He has published broadly in top-tier security (IEEE S&P, USENIX Security, CCS, IEEE TIFS, and IEEE TDSC) and software engineering venues including ICSE, FSE, ASE, ACM TOSEM, and IEEE TSE. More information is available on https://sen-chen.github.io/.

Lingling Fan is an Associate Professor in College of Cyber Science, Nankai University, China. She received her Ph.D and BEng degrees in computer science from East China Normal University, Shanghai, China in June 2019 and June 2014, respectively. In 2017, she joined Nanyang Technological University (NTU), Singapore as a Research Assistant and then had been as a Research Fellow of NTU since 2019. Her research focuses on program analysis and testing, software security. She got two ACM SIGSOFT Distinguished Paper Awards at ICSE 2018.

Chen Chunyang obtained his Ph.D. degree from School of Computer Science and Engineering, Nanyang Technological University (NTU), Singapore, and bachelor's degree from Beijing University of Posts and Telecommunications (BUPT), China, June 2014. He is a lecturer (a.k.a. Assistant Professor) in Faculty of Information Technology, Monash University, Australia. His research focuses on Mining Software Repositories, Text Mining, Deep Learning, and Human Computer Interaction.

Liu Yang graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2011, Dr. Liu is awarded the Temasek Research Fellowship at NUS to be the Principal Investigator in the area of Cyber Security. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently a full professor and the director of the cybersecurity lab in NTU.

He specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. His work led to the development of a state-of-the-art model checker, Process Analysis Toolkit (PAT). By now, he has more than 300 publications and 6 best paper awards in top tier conferences and journals. With more than 20 million Singapore dollar funding support, he is leading a large research team working on the state-of-the-art software engineering and cybersecurity problems.