Aladdin: Automating Release of Android Deep Links to In-App Content

Yun Ma, Xuanzhe Liu, Ziniu Hu, Dian Yang, Gang Huang, Yunxin Liu, Tao Xie

Abstract

Unlike the Web where each web page has a global URL to reach, a specific "content page" inside a mobile app cannot be opened unless the user explores the app with several operations from the landing page. Recently, deep links have been advocated by major companies to enable targeting and opening a specific page of an app externally with an accessible uniform resource identifier (URI). To empirically investigate the state of the practice on adopting deep links, in this paper, we present the largest empirical study of deep links over 20,000 Android apps, and find that deep links do not get wide adoption among current Android apps, and non-trivial manual efforts are required for app developers to support deep links. To address such an issue, we propose the Aladdin approach and supporting tool to release deep links to access arbitrary location of existing apps. Aladdin instantiates our novel cooperative framework to synergically combine static analysis and dynamic analysis while minimally engaging developers to provide inputs to the framework for automation, without requiring any coding efforts or additional deployment efforts. We evaluate Aladdin with popular apps and demonstrate its effectiveness and performance.

1 Introduction

One significant feature of the Web is that there are zillions of hyperlinks to access web pages and even to a specific piece of "*deep*" web content. For example, the hyperlink `https://en.wikipedia.org/wiki/World_Wide_Web#History` points to the "*history*" anchor location of the "*WorldWideWeb*" wiki page. With hyperlinks, the users can directly navigate to arbitrary locations on the Web with just one click. Indeed, the hyperlinks play a key role on the Web such as navigation among web pages and bookmarks, and hyperlinks are crucial for search engines to crawl the web contents [16].

In the current era of mobile computing, apps have been the dominant entrance to access the Internet rather than web pages. However, mobile apps historically lack the consideration of hyperlinks. At the early age of mobile apps, developers focus on only the features and functionalities, without having strong motivations to provide hyperlink-like support to a specific in-app content. Accessing a specific in-app content requires users to launch this app and land on the "home" page, locate the page/location containing the content by a series of actions such as search-and-tap and copy-and-paste, and finally reach the target. Compared to the Web, the support such as "hyperlinks" is inherently missing so that users have to perform tedious and trivial actions. Other benefits from traditional web hyperlinks are naturally missing as well.

Realizing such a limitation, the concept of "**Deep Link**" has been proposed to enable directly opening a specific page/location inside an app from the outside of this app by means of a uniform resource identifier (URI) [8]. Intuitively, deep links are "hyperlinks" for mobile apps. For example, with the deep link "`android-app://org.wikipedia/http/en.m.wikipedia.org/wiki/World_Wide_Web`", users can directly open the page of "*WorldWideWeb*" in the Wikipedia app. Currently, Google [7], Facebook [6], Baidu [3], and Microsoft [4] strongly advocate the concept of deep links, and major mobile OS platforms such as iOS [14] and Android [2] encourage their developers to release deep links. With deep links, mobile users can automatically navigate from one app to a specific page/location of other apps installed on her device, without manually switching apps.

Indeed, deep links bring various benefits to current mobile users, app developers, and service providers. However, it is unclear how deep links have been supported so far in the state of the broad practice. To address such a question, in this paper, we first present an empirical study to investigate (1) the trend of deep links from version evolutions among 200 popular Android apps; (2) the coverage of deep links among 20,000 popular apps; and (3)

developers' coding efforts to support deep links on 8 open-source apps from GitHub. We find that the current deep links (1) have ***low coverage*** among Android apps, i.e., more than 73% of the apps do not have any deep links; (2) require ***non-trivial developer manual efforts*** to implement, i.e., developers need to manually modify 45–411 lines of code per deep link.

Essentially, releasing deep links for apps is to support programmable execution of apps to a specific location. However, manually implementing programmable execution to all the locations inside an app is not possible due to the high complexity of current apps. Thus the coverage is very low so far. To improve the coverage with minimal developer efforts, one possible solution is to leverage program analysis of apps to extract execution traces. However, static analysis cannot reach the dynamic contents generated at runtime, and the state-of-the-art test generation tools for Android apps suffer from achieving low code coverage [22].

To address such a challenge, in this paper, we propose ***Aladdin***, a novel approach that can help developers automate the release of deep links for Android apps based on a cooperative framework. Our cooperative framework combines static analysis and dynamic analysis as well as engaging minimal human efforts to provide inputs to the framework for automation. In particular, given the source code of an app, Aladdin first uses static analysis to find how to reach activities (each of which is the basic UI component of Android apps) inside an app most efficiently from the entrance of the app. After developers select activities where deep links to dynamic locations are needed, Aladdin then performs dynamic analysis to find how to reach fragments (each of which is a part of a UI component of Android apps) inside each activity. Finally, Aladdin synthesizes the analysis results and generates the templates that record the scripts of how to reach arbitrary locations inside the app. Aladdin provides a deep-link proxy integrated with the source code of the app to take over the deep-link processing, and thus does not instrument the original business logic of the app. Such a proxy can accept the access via released deep links from third-party apps or services. We evaluate Aladdin on 20 popular Android apps. The evaluation results show that Aladdin can cover a large portion of an app, and the runtime overhead is quite marginal.

This paper makes the following contributions.

- We conduct an extensive empirical study of current deep links based on 20,000 popular Android apps and identify the factors preventing the prevalence of deep links.
- We propose an approach to automating the release of deep links based on a cooperative framework, with only minimal developers' configuration efforts and no interference of the normal functionalities of an app.
- We evaluate the feasibility and efficiency of our approach on popular Android apps.

The rest of this paper is organized as follows. Section 2 describes the background of Android apps and deep links. Section 3 presents the empirical study of current deep links. Section 4 presents our approach to automating the release of deep links. Section 5 presents the implementation details. Section 6 illustrates the evaluations. Section 7 discusses some issues and extensions of our approach. Section 8 highlights related work and Section 9 concludes the paper.

2 Background

Essentially, deep links enable the direct access to a location inside an app, just like hyperlinks enabling the direct access to a location on a webpage. Before introducing the empirical study and Aladdin approach, we first present some background knowledge of deep links in Android apps.

2.1 Android Apps

Figure 1 shows the structure of a typical Android app [1]. An app, identified by its `packageName`, usually consists of multiple `activities` that are loosely bound to each other. An `activity` is a component that provides a user interface with which users can interact and perform some tasks, such as dialing phones, watching videos, reading news, or viewing maps. Each `activity` has a unique `className` and is assigned a window to draw its graphical user interface. One `activity` in an app is specified as the "main" activity, which is first presented to the user when the app is launched.

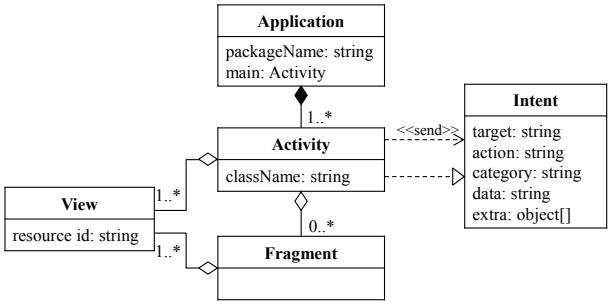


Figure 1: Android application model.

For ease of understanding, we can draw an analogy between Android apps and the Web, as compared in Table 1. An Android app can be regarded as a website where the packageName of the app is like the domain of the website. Therefore, activities can be regarded as web pages because both of them are basic blocks for apps and websites, respectively, providing user interfaces for users to interact with. The main activity is just like the home page of a website.

An activity has several `Views` to display its user interface, such as `TextView, ButtonView, ListView`. Views are similar to web elements consisting of a web page, such as <p>, <button>, and <ul>. When a web page is complex, frames are often used to embed some web elements for better organization. Frames are subpages of a web page. Similarly, since the screen size of mobile devices is rather limited, Android provides `Fragment` as a portion of user interfaces in an activity. Each fragment forms a subpage of an activity.

Activities and fragments hold the contents inside apps, just like web pages and frames encapsulate contents on the Web. In the rest of this paper, we use the term "page" and "activity" exchangeably, as well as "subpage" and "fragment" exchangeably, for ease of presentation.

Activities are transited through `Intents`. An Intent is a messaging object used to request an action from another component, and thus essentially supports Inter-Process Communication (IPC) at the OS level. There are two types of intent: (1) an *explicit* intent, which specifies the target activity to start by the `className`; (2) an *implicit* intent, which declares `action`, `category`, and/or `data` that can be handled by another activity. Messages are encapsulated in the `extra` field of an intent. When an activity P sends out an intent I, the Android system finds the target activity Q that can handle I, and then loads Q, achieving the transition from P to Q.

Table 1: Conceptual Comparison between Web and Android Apps.

Concepts of Android Apps	Concepts of Web
app	website
packageName	domain
activity	web page
main activity	home page
view	web element
fragment	frame

2.2 Deep Links

The idea of deep links for mobile apps originates from the hyperlinks in the Web. One significant difference between the Web and mobile apps is that each web page has a globally unique identifier URL. In this way, web pages are accessible directly from anywhere by means of a URL. Typically, web users can enter the URL of a web page in the address bar of web browsers, and click the "Go" button to open the target web page. They can also click through hyperlinks on one web page to navigate directly to other web pages.

Compared to web pages, the mechanism of hyperlinks is historically missing for mobile apps. A page of an app has only an internal location. Reaching a page in an app has to be started by launching the app, and users may need to walk through various pages to reach the target one. For example, a user can find a favorite restaurant in a food app such as `Yelp`. Next time when the user wants to check the restaurant information, she has to launch the app again, search the restaurant again, and then reach the page of the restaurant's details. In other words, there is no way for the user to directly open the restaurant page even if she has visited before.

To solve the problem, deep links are proposed to enable directly opening a specific page inside an app from the outside of this app with a uniform resource identifier (URI). The biggest benefit of deep links is not limited in enabling users to directly navigate to a specific location of an app with a dedicated link, but further supports other apps or services (e.g., search engines) to be capable of accessing the internal data of an app and thus enables "communication" of apps to explore more features, user experiences, and even revenues.

In practice, deep links are implemented with implicit intent. The first step is to add a specific kind of intent filters for activities displaying the desirable pages. In the `AndroidManifest.xml`, intent filters declare the URI patterns that the app handles from inbound links. The code snippet in Listing 1 illustrates an example of intent filter for links pointing to `http://www.examplepetstore.com`:

Listing 1: Intent-Filter Configuration of Deep Links

```
<activity android:name="com.example.android.PetstoreActivity"
android:label="@string/title_petstore">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http" />
    <data android:host="www.examplepetstore.com" />
  </intent-filter>
</activity>
```

The second step is to add the business logic to handle the intent filters. The code snippet in Listing 2 shows the logic of handling the intent filter defined previously. Once the Android system starts the app activity through an intent filter, the business logic is called to use the data provided by the intent to determine the app's view response. The methods `getDataString`() can be used to retrieve the data associated with the incoming intent during the launching callbacks of the activity such as `onCreate`() or `onStart`().

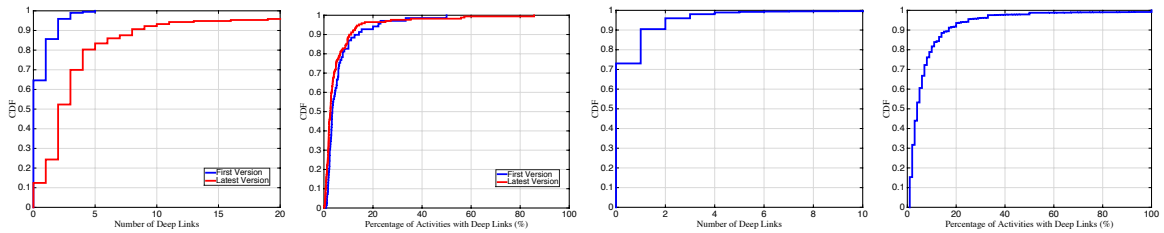
Listing 2: The logic of handling the intent filter

```
protected void onNewIntent(Intent intent) {
   String action = intent.getAction();
   String data = intent.getDataString();
   if (Intent.ACTION_VIEW.equals(action) && data != null) {
      String productId = data.substring(data.lastIndexOf("/") + 1);
      Uri contentUri = PetstoreContentProvider
               .CONTENT_URI.buildUpon()
               .appendPath(productId).build();
      showItem(contentUri);
   }
}
```

3 Empirical Study of Deep Links

Given the benefits of deep links for mobile apps, in this section, we present an empirical study to understand the state of practice of deep links in current Android apps. We focus on three aspects: *(1) the trend of deep links with version evolution of apps*; *(2) the number of deep links in popular apps*; (3) *how deep links are realized in current Android apps.*

In practice, there is no standard way of measuring how many deep links an app has. However, from Section 2.2, we can infer an essential condition, i.e., ***activities that support deep links MUST have a special kind of intent filters declared in the `AndroidManifest.xml` file***. Such a kind of intent filters should use the `android.intent.action.VIEW` with the `android.intent.category.BROWSABLE` category. We denote these intent filters as deep-link related.



(a) Trend of apps with deep links (b) Trend of deep-linked activities in deep-link supported apps (c) Distribution of the number of deep links (d) Distribution of the percentage of deep-linked activities

Figure 2: The trend and status of deep links among Android apps.

Therefore, we can simply take the number of activities with deep-link related intent filters as an indicator to estimate the number of deep links for an Android app. In other words, if an activity has a deep-link related intent filter, we regard that such an activity has a deep link.

3.1 Evolution of Deep Links with Versions

We first validate that the support of deep links is really desired. To this end, we investigate the trend of deep links along with the version evolution. Due to the banned Internet access in mainland China, we cannot perform large-scale study on apps on Google Play. Instead, we choose top 200 apps ranked by Wandoujia[1], a leading Android app store in China which was studied in our previous work [25, 26]. To make comparison, we crawl each app's first version that could be publicly found and its latest version published on its official website as of August 2016. We compare the number of deep links of the two versions of each app. Figure 2(a) indicates the change of the number of deep links across these two versions. Generally, it is observed that when the app is first released on the app store, only about 35% of apps have deep links. In contrast, more than 87% of these apps have supported deep links in their latest versions. More specifically, the maximum number of deep links is only 5 in the first version of all investigated apps (the app with the most deep links is *Tencent News*). In contrast, 20% of the investigated apps have more than 5 deep links in their latest versions, and four apps (*Dianping*, *Meituan*, *Taobao*, *Sina Weibo*) each have even more than 100 deep links. Such a change indicates the popularity of deep links keeps increasing in the past few years.

3.2 Coverage of Deep Links

Although the number of deep-link supported apps increases, to our surprise, the percentage of activities that have deep links in deep-link supported apps is still rather low. We compute the ratio of deep-link supported activities relative to the total number of activities. As shown in Figure 2(b), the distribution of the percentage does not change much between the first version and latest version. Among 90% of deep-link supported apps, only less than 10% of activities have deep links. A small difference is that the maximum percentage of deep-linked activities increases from 50% in the first version to 85% in the latest version.

Aiming to expand the investigation to a wide scope, we study the latest version of top 20,000 apps ranked by the number of downloads on Wandoujia as of August 2016. Figure 2(c) demonstrates the coverage of deep links among these apps. Similar to the preceding results, about 73% of the apps do not have deep links, while 18% of the apps have only one deep link. Such a result indicates that deep links are not well supported in a large scope of current Android apps.

Considering the percentage of deep-linked activities in deep-link supported apps, Figure 2(d) shows that the median percentage is just about 5%, implying that a very small fraction of the pages can be actually accessed via deep links. Considering the best cases, only 5% of apps have more than 20% of their activities support deep links. In summary, the preceding empirical study demonstrates the low coverage of deep links in current Android apps.

[1] http://www.wandoujia.com

Table 2: LoC changes when adding deep links of open-source apps on GitHub.

Repository Name	LoC Changes
stm-sdk-android	45
mobiledeeplinking-android	62
WordPress-Android	73
mopub-android-sdk	78
ello-android	87
bachamada	179
sakai	237
SafetyPongAndroid	411

Such result is a bit out of our expectation, since deep links are widely encouraged in industry to facilitate in-app content access.

3.3 Developer Effort

The preceding empirical results indicate that the coverage of deep links is not satisfying. Indeed, supporting deep links requires the developers to write code and implement the processing logics. In practice, there are usually two ways of implementing deep links. One is to establish implicit intents for each activity that needs to be equipped with deep links. The other is to use a central activity to handle all the deep links and dispatch each deep link to its target activity. Although there have already been some SDKs for deep links [9, 5], implementing deep links exactly requires the modifications or even refactoring of the original implementation of the apps. We then investigate the actual developer effort when releasing deep links for an Android app.

For simplicity, we study the code evolution history of open-source apps on GitHub. We search on GitHub with the key word "*deep link*" among all the code-merging requests in the Java language. There are totally 4,514 results returned. After manually examining the results, we find 8 projects that actually add deep links in their code-commit history. Table 2 shows the LoC (lines of code) of changes in each project when adding ***one*** deep link. The changes include the addition, modification, and deletion of the LoC. We can observe that the least number of the LoC is 45 and the biggest can reach 411. For instance, `Wordpress` app adds a deep link to its sign-in activity with 73 LoC changes. Adding the intent-filter has only 22 LoC changes, but dealing with a ąřprefillUrląś variable changes 51 LoC in 3 files. For the app `SafetyPongAndroid`[2], we find that a large number of changes attribute to the refactoring of app logics to enable an activity to be directly launched without relying on other activities. Such an observation can provide us the preliminary findings that developers need to invest non-trivial manual efforts on existing apps to support deep links. Such a factor could be a potential reason why deep links are of low coverage.

4 Aladdin: In a Nutshell

The findings of our empirical study demonstrate the **low coverage** and **non-trivial developer efforts** of supporting deep links in current Android apps. To improve coverage, one possible solution is to leverage the program analysis of apps to extract how to reach locations of in-app contents pointed by deep links. However, the static analysis of Android apps can only build structure relations among activities but cannot analyze dynamic fragments inside an activity; the dynamic analysis of Android apps can analyze dynamic fragments but suffer from low coverage of activities, i.e., only a small fraction of activities can be reached by dynamic analysis.

To address the challenge, we propose a cooperative framework and design a tool ***Aladdin*** to automate the release of deep links. Our cooperative framework combines static analysis and dynamic analysis while minimally engaging developers to provide inputs to the framework for automation. Figure 3 shows the overview of our approach, and the workflow is illustrated as follows.

Given the source code of an app, Aladdin first derives deep link templates that record the scripts of how to reach arbitrary locations inside the app. Each template is associated with an activity and consists of two parts: one

[2] `https://github.com/SafetyMarcus/SafetyPongAndroid`

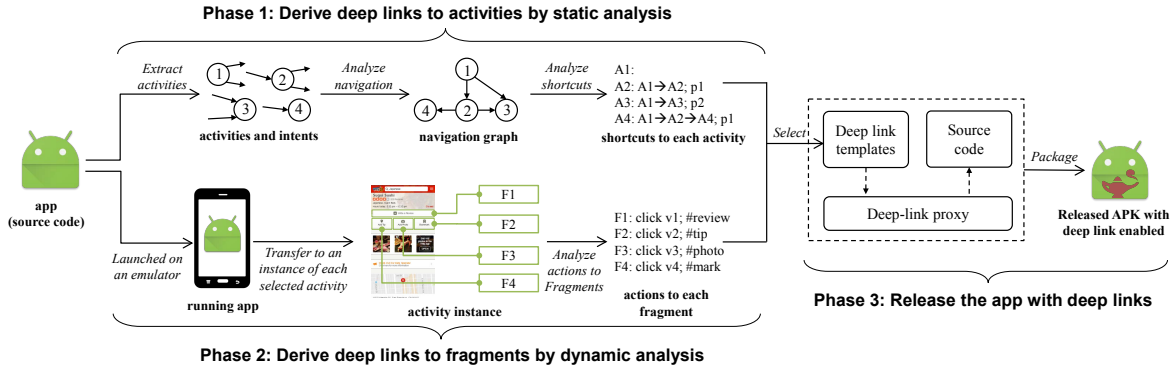


Figure 3: Approach Overview.

part is the intent sequence extracted based on static analysis, recording how to reach the activity from the main activity of the app; the other part is the action sequence extracted based on dynamic analysis, recording how to reach fragments in the activity from the entrance of the activity. After developers configure to verify the activities and fragments to be deep linked, the corresponding templates and a deep-link proxy are automatically generated, and are packaged with the original source code of the app as a new `.apk` file. Each template has a URI schema that is used to populate a concrete deep link. The deep-link proxy provides a replay engine that can replay the sequences at runtime to execute the deep links.

When a deep link is requested with a URI conforming to a schema, the deep-link proxy is triggered to work. The corresponding template is instantiated by assigning values to parameters in the template. Then the proxy first makes the app transfer to the target activity by issuing the intents one by one in the intent sequence. Then the proxy makes the activity transfer to the target fragment by performing the actions one by one in the action sequence. Finally, the target location could be reached. Such a proxy-based architecture does not modify any original business logic of the app and is coding-free for developers.

We next present the details of each phase.

4.1 Deriving Deep Links to Activities

Essentially, reaching an activity with a deep link is to issue one intent that could launch the target activity. However, the complexity of activity communications in Android apps makes it not easy to use a single intent to reach an activity for existing apps. For example, a target activity may rely on internal objects that are created by previous activities in the path from the main activity to the target activity. To address the issue, rather than the single intent to an activity, Aladdin abstracts a deep link to an activity as a sequential path of activity transitions via intents, starting from the main activity of the app to the target activity pointed by the deep link. Therefore, the activity dependency can be resolved because we actually follow the original logic of activity communications.

4.1.1 Navigation Analysis

Since activities are loosely coupled that are communicated through intents, there is no explicit control flows between activities. Therefore, we design a *Navigation Graph* to abstract the activity transitions. Here we give the formal definitions of activity transition and navigation graph.

Definition 1 (Activity Transition)**.** *An activity transition $t(\mathscr{L})$ is triggered by an intent, where $\mathscr{L}$ is the combination of all the fields of the intent including action, category, data, and objects of basic types from the extra field.*

Since an intent essentially encapsulates several messages passed between two activities, we use a label set $\mathscr{L}$ to abstract an intent. Two intents are equivalent if and only if the label sets are completely the same. Note that, from

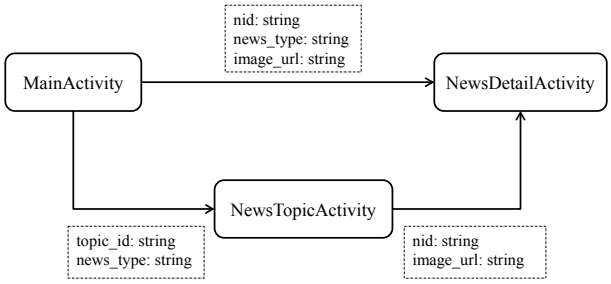


Figure 4: Example of Wallstreet News App.

the extra field which is the major place to encapsulate messages, we take into account only the objects of basic types including `int`, `double`, `String`, etc. The reason is that objects of app-specific classes are usually internally created but cannot be populated from outside of the app. As a result, this kind of intents cannot be replayed at runtime.

Definition 2 (Navigation Graph)**.** *A Navigation Graph G is a directed graph with a start vertex. It is denoted as a 3-tuple, $G < V, E, r >$, where V is the set of vertices, representing all the activities of an app; E is the set of directed edges, and every single $e(v_1, v_2)$ represents an activity transition $t(\mathscr{L})$; r is the start vertex.*

In such a navigation graph, the start vertex r refers to the main activity of the app. We assume that each node in V could be reachable from the start vertex r. The navigation graph can have multi-edges, i.e., $\exists e_1, e_2 \in E, v_{start}(e_1) = v_{start}(e_2)$ *and* $v_{end}(e_1) = v_{end}(e_2)$, indicating that there can be more than one transition between two activities. In addition, it should be noted that the navigation graph can be cyclic.

4.1.2 Shortcut Analysis

After constructing the navigation graph, we can analyze the paths to each activity.

Definition 3 (Path)**.** *A path to an activity $\mathscr{P}_a$ is an ordered list of activity transitions $\{t_1, t_2, \ldots, t_k\}$ starting from the main activity, where k is the length of the path.*

According to the path definition, the activity transition t_1 is always the app-launching intent that opens the main activity. The path $\mathscr{P}_a$ can ensure that all the internal dependencies are properly initialized before reaching the activity a.

In practice, there can be various paths to reach a specific activity. Figure 4 shows the example in the "*Wallstreet News*" app. If we want to reach a news page (`NewsDetailActivity`), there are two paths. One is to navigate directly from the `MainActivity`. The other is to switch to the topic page of `NewsTopicActivity` and then navigate to the news page. Obviously, the former path is shorter than the latter one. Since our approach uses the activity transitions to reach activities by deep links, the path should be as short as possible to reduce the execution time at system level. However, in some circumstances, a shorter path to an activity cannot cover all the instances of the activity. For example, the intermediate activities on the path can depend on some internal variables, so that they can be reached only via a longer path where these variables are assigned.

We define that the path p_1 can replace the path p_2 if and only if $\mathscr{L}_{p_1} \subset \mathscr{L}_{p_2}$. Here $\mathscr{L}_{p_j}$ is the combination of all the labels in the path p_j: $\mathscr{L}_{p_j} = \cup\{\mathscr{L}(t_i) | t_i \in p_j\}$. For example, in the "*Wallstreet News*" app, the path 1 from the `MainActivity` to the `NewsDetailActivity` has three labels, i.e., "*nid*", "*image_url*", and "*news_type*". We can observe that all these three labels are included in path 2 which passes the `NewsTopicActivity`. Hence, we can use path 1 to replace path 2.

With the definition of path replacement, we can define the possible shortest path as a Shortcut. We can use the shortcut to replace the path to an activity accounting for shorter execution time.

Definition 4 (Shortcut)**.** *A Shortcut of a path $\mathscr{T}(p)$ is the shortest path that can replace path p.*

We design the Algorithm 4.1 for finding the shortcuts in navigation graph G. For every vertex in the graph, we get its paths (Line 2) and sort the paths by their length in ascending order (Line 3). Then we enumerate every path in the path list, to find the shortest path that can replace it (Line 4-12), if they fulfill the requirement of label containing (Line 7). Due to the increasing sequence, the first available path that we obtain is the shortest one. We store it in a two-dimension map structure (Line 8).

```
   Input: Navigation Graph G < V, E, r >
   Output: Shortcut 𝒞
 1 foreach v ∈ V do
 2 |   Plist ← G.path(v);
 3 |   sort_by_length(Plist);
 4 |   for i ← 1 to Plist.length do
 5 |   |   Shorcut[< v, p_i >] ← p_i;
 6 |   |   for j ← 1 to i − 1 do
 7 |   |   |   if ℒ_{p_j} ⊂ ℒ_{p_i} then
 8 |   |   |   |   Shorcut[< v, p_i >] ← p_j;
 9 |   |   |   |   break;
10 |   |   |   end
11 |   |   end
12 |   end
13 end
```

Algorithm 4.1: Shortcut computation.

After computing the shortcuts to an activity, we filter out the unique shortcuts and then put all of them into the intent sequence in the deep link template corresponding to the activity. The labels in the intents can be configured by developers as parameters. At runtime, these parameters are assigned values to reconstruct the shortcut.

4.2 Deriving Deep Links to Fragments

As shown in Section 2, there are different fragments in an activity served as user interface, just like the frames in a web page. In order to reach a specific fragment directly with a deep link, we should further analyze how to transfer to fragments of an activity.

Contrary to activity transitions where intents can be sent to invoke the transition, the fragment transitions often occur after users perform an action on the interface such as clicking a view, then the app gets the user action and executes the transition. Due to the dynamics of activities, fragments of activities may be dynamically generated, just like AJAX on the Web. To the best of our knowledge, it is currently not possible to find out fragments by static analysis. Thus, we tend to use dynamic analysis, traversing the activity dynamically by clicking all the views on the page in order to identify all the fragments and their corresponding trigger actions.

4.2.1 Fragment Identification

Unlike activities where className is the identifier of different activities, fragments usually do not have explicit identifies. To determine whether we have switched the fragment after clicking a view, we use the view structure to identify a certain fragment. In Android, all the views are organized in a hierarchy view tree. We get the view tree at runtime and design Algorithm 4.2 to calculate the structure hash of this tree, and use the hash to identify the fragments. The algorithm is recursive with a view r as input. If r does not have children, the result is only the string hash of r's view tag (Line 2). If r has children (Line 3), then we use the algorithm to calculate all the hash of its children recursively (Lines 5-7). Then, we sort the r's children based on their hash values to ensure the consistency of the structure hash, because a view's children do not keep the same order every time(Line 8). Next, we add each children's hash together with the view tag forming a new string (Line 10), and finally return the string hash (Line 13). When inputing the root view of the tree to the algorithm, we could get a structure hash of the view tree. The hash can be used as an identifier of a fragment.

```
   Input: View r
   Output: Structure Hash h
1  function TreeHash(r)
2  str ← r.viewTag
3  if r.hashChildren() then
4  |   children ← r.getChildren()
5  |   foreach c ∈ children do
6  |   |   c.hash ← TreeHash(c)
7  |   end
8  |   children ← sort_by_hash(r.getChildren())
9  |   foreach c ∈ children do
10 |   |   str+ = c.hash
11 |   end
12 end
13 return hash(str)
```

Algorithm 4.2: Computing structure hash of view tree.

4.2.2 Fragment Transition Graph

In order to retrieve all the fragments as well as triggering actions to each fragment, we define a fragment transition graph to represent how fragments are switched in an activity.

Definition 5 (Fragment Transition Graph)**.** *A Fragment Transition Graph is a directed graph with a start vertex. It is denoted by a 3-tuple, $FTG < V, E, r >$. V is the set of vertices, representing all the fragments of an activity, identified by the structure hash. E is the set of directed edges. Each edge e is a 3-tuple, $e < S, T, I >$. S and T are source and target fragments where I represents the resource id of the view which invokes the transition. r is the start vertex.*

The dynamic analysis performs on an instance of an activity. Therefore, after developers select the activity to support deep links to fragments, a simulator will be launched and developers are asked to transfer to an instance page of this activity. From this page, the simulator traverses the activity in the depth-first sequence as the algorithm 4.3 describes. For each view in the current fragment, we try to click it (Lines 2-3) and check whether the current state has changed. If the activity has changed, then we can use the system function `doback()` to directly return to previous condition (Line 5). Otherwise, we check the fragment state. If the structure hash of the current fragment is different from that of the previous one, the fragment has changed (Lines 8-9). Therefore, we can add it into the edge set if it is a new fragment (Line 10-12). The dynamic analysis is similar to the web crawlers that need to traverse every web page, except that Android only provides a `doback()` method that can return to the previous activity, but not to the previous fragment. So to implement backtrace after fragment transitions, we have to restart the app and recover to the previous fragment (Line 14).

After finishing the traverse search, we can get the fragment transition graph and a list of fragments. To get the path towards a certain fragment, we simply combine all the edges from the start vertex to the fragment. Thus, developers can choose any fragment to form a deep link by putting the actions into the action sequence in the deep link template.

4.3 Releasing Deep-Link Supported Apps

After computing the shortcuts to activities and actions to fragments, the next step is to create the target `.apk` that supports processing deep links at runtime. Note that developers may want to create deep links to only some locations of their apps. Therefore, Aladdin allows developers to check which ones need the support of deep links.

Then Aladdin generates an abstract URI of the deep links for each selected activity. To be discovered by Android system, the URI should follow the format of "*scheme://host/path*" where *scheme*, *host* and *path* could be any string value. In order to conform to the latest app links specification of Android 6 [2], we employ the format of "*http://host/target? parameter#fragment*" as the schema of the abstract URI. We use the reverse string

```
   Input: Activity a, Fragment f
   Output: Fragment Transition Graph G
1  function FTGBuild(a, f)
2  foreach v ∈ f.views() do
3      v.click()
4      if getCurrentActivity() ≠ a then
5          doback()
6          continue
7      end
8      cf ← getCurrentFragment()
9      if TreeHash(cf.root)≠TreeHash(f.root) then
10         if cf ∉ G.V then
11             G.E.add(< f, cf, v >)
12             FTGBuild(a, cf)
13         end
14         recover()
15         continue
16     end
17 end
```

Algorithm 4.3: Generation of the Fragment Transition Graph.

of the `packageName` (usually the domain of the corresponding website) for the *host* field and the `className` of the activity for the *target* field. All the instances of an activity share the similar prefix before "?" but are different on the *parameter* part. For deep links to fragments, the name of the target fragment is after a #. With the abstract URI, we can generate intent filters in AndroidManifest.xml file to handle the corresponding deep links.

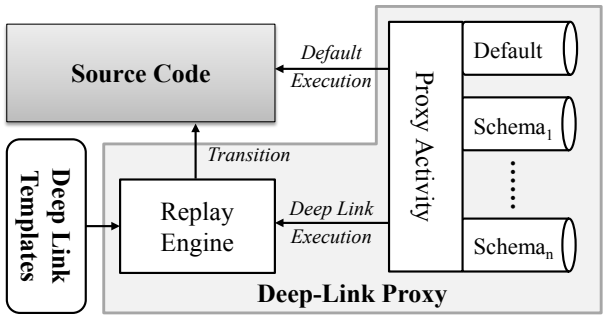


Figure 5: Structure of the app with deep link enabled.

Figure 5 depicts the structure of the created `.apk`. We leverage a proxy architecture to realize minimal refactorings to the original app. A *Proxy Activity* is used to handle all the incoming requests. The *Proxy Activity* is configured to intent filters that conform to the URI schemas. When an intent is passed to the *Proxy Activity*, if the intent matches one of the schemas, the *Proxy Activity* informs the *Replay Engine* to execute the deep link. If the incoming intent cannot match to any of the schemas, it is then forwarded directly to the original *Source Code* for default execution.

Figure 6 shows two deep link templates of the `Anki` app released by Aladdin. The two pieces of code in the box are deep link templates for `CardTemplateEditor` and `NoteEditor` activity. Two intents with two parameters `CALLER` and `modeId` have to be issued before reaching `CardTemplateEditor`. Therefore, the deep link to `CardTemplateEditor` should explicitly specify the values of the two parameters. Then Aladdin can populate the proper intents to transfer to the target activity. `NoteEditor` has a fragment naming "*tags*". The actions to the fragment is clicking the view whose resource id is `CardEditorTagButton`. Therefore, to reach the tags fragment, not

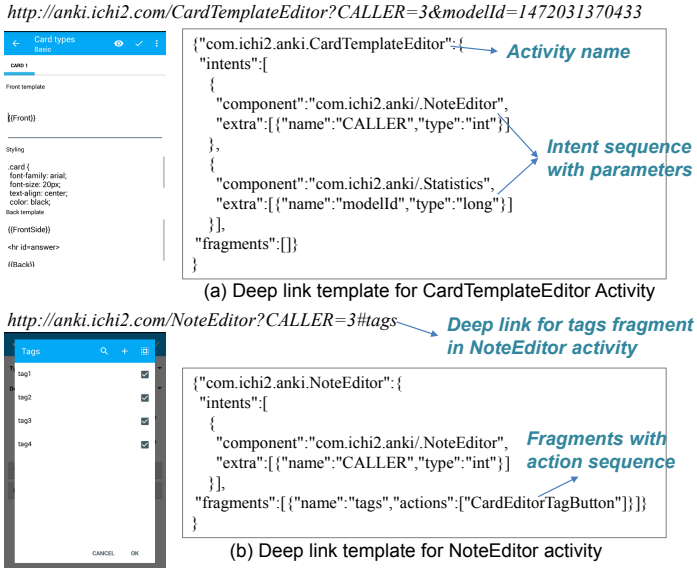


Figure 6: Example of deep link templates and concrete deep links.

only should the value of intents be assigned ($CALLER = 3$), but the fragment should be specified as well ($\#tags$).

4.4 Executing Deep Links by Replay at Runtime

When the deep link is executed, the corresponding deep-link template is instantiated with concrete values to create a replay script. Then the *Replay Engine* communicates with the original *Source Code* and instructs the app to transit through activities and perform actions on views according to the script.

For example, in Figure 6(b), when the deep link `http://anki.ichi2.com/NoteEditor?CALLER=3#tags` is requested, it implies that the user may want to reach the tags fragment of *NoteEditor* in `Anki app`. So the Replay Engine first issues the intent with the parameter `CALLER` is 3. Then the Replay Engine performs a click on the view whose resource id is `CardEditorTagButton`. Finally, the user can reach the target location.

5 Implementation

In this section, we present the implementation details of Aladdin.

To construct the Navigation Graph, we first use the `IC3` tool [29] to extract all the activities and intents from the source code of an Android app. Then we apply the `PRIMO` tool [28] to compute the links among activities. For each link computed by `PRIMO`, we add the corresponding activity as a node to the *Navigation Graph* and connects the two nodes with an edge. The labels on the edge are retrieved from the output of `IC3`. In particular, some edges have only a sink activity, indicating that this activity can be directly opened from outside of the app. For these edges, we add an edge from the main activity node, to make all the nodes reachable from the main activity.

We use the instrumentation test framework provided by Android SDK to implement the dynamic analysis. Android instrumentation can load both a test package and the **App-Under-Test** (AUT) into the same process. With this framework, we are able to inspect and change the runtime of an app, such as retrieving view components of an activity and triggering a user action on the target app.

To generate the `.apk` with deep link enabled, the *Replay Engine* is actually implemented as a test case of the instrumentation test to execute the scripts. The deep-link proxy is implemented as a normal Android activity and

the corresponding schemas are regularly configured as intent-filters in the `AndroidManifest.xml` file. When the proxy activity receives a deep link request, it launches the instrumentation test to run the Replay Engine.

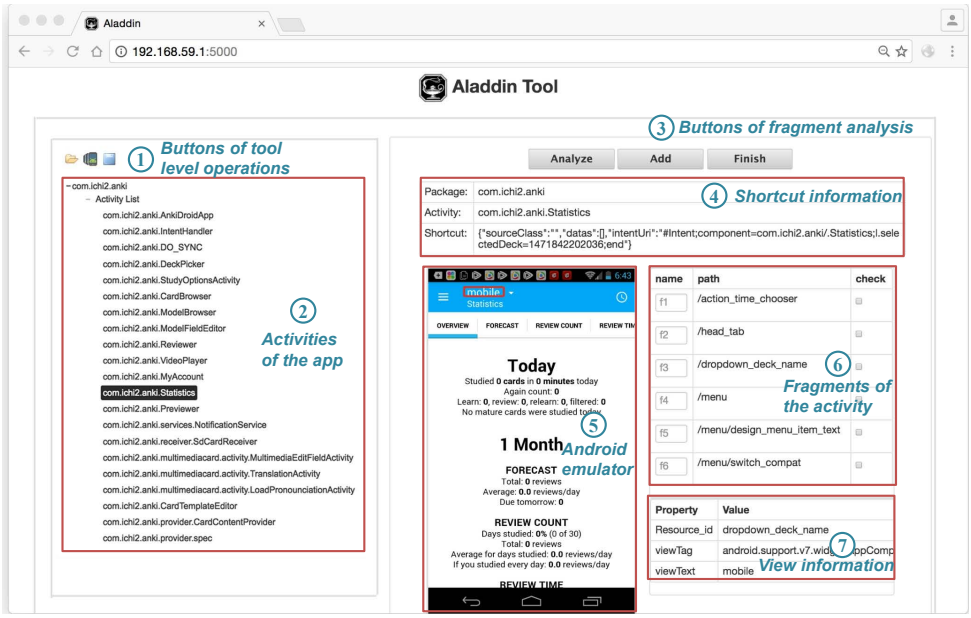


Figure 7: Snapshot of Aladdin tool.

Figure 7 shows the snapshot of Aladdin tool. Developers can choose the source code of apps to be analyzed from the button in ①. Then the static analysis is automatically performed to extract all the activities (shown in ②) as well as the shortcuts of each activity. When an activity is selected, the shortcut information is shown in ④. If a deep link to fragments is required, Aladdin allows developers to launch an emulator from ①, then transfer to an instance of the activity by performing actions in ⑤. Next, if the dynamic analysis is chosen to run from ③, the executed results are listed in ⑥. Developers can explore whether each fragment is correct by clicking each result in ⑤. Developers can also manually add a fragment by performing the actions in ⑤ with view information in ⑦. After selecting all the fragments that need deep links, developers click finish button from ③ to complete the analysis for an activity. When the deep links for activities have been confirmed, developers can click the "Save" button in ① and package the `.apk` file as output.

6 Evaluations

In this section, we evaluate the feasibility and efficiency of our Aladdin approach. Our evaluations address the following research questions:

- **RQ1**: How high coverage can Aladdin achieve?
- **RQ2**: How accurate is Aladdin's dynamic analysis of fragments?
- **RQ3**: How high runtime overhead does Aladdin's proxy activity incur?

6.1 App Dataset

We evaluate Aladdin on top apps chosen from Google Play. Aladdin is applied on the source code of an Android app. However, most of top apps on Google Play are not open source. Although there are some disassembly tools that can generate Java-like code from the `.apk` file, many apps still fail to be analyzed by Aladdin due

Table 3: Selected apps for evaluations.

App	Category	Description	Downloads	# of Activities	Current # of Deep Links	# of Deep Links by Aladdin	Increase of Coverage
Cool Wallpapers	Personalization	Wallpaper app	5M-10M	9	0	4	44%
Fitness & Bodybuilding	Health & Fitness	Fitness & Bodybuilding	1M-5M	9	0	4	44%
NFL Mobile	Sports	Football app	10M-50M	23	1	12	48%
Wikipedia	Books & Reference	Wikipedia client	500k-1M	18	1	11	56%
NPR News	News & Magazines	News reader	1M-5M	17	0	17	100%
AnkiDroid Flashcards	Education	Flash card manager	1M-5M	21	1	14	62%
Lyft Taxi	Transportation	Taxi service	1M-5M	12	2	6	33%
Booking.com Hotel Reservations	Travel & Local	Hotel reservations	10M-50M	120	4	112	90%
APUS Booster+	Productivity	System utility	10M-50M	20	0	7	35%
Hulu	Entertainment	Live streaming	100K-500K	1	0	1	100%
Music Player for Android	Music & Audio	Music player	10M-50M	13	0	5	38%
Marvel Comics	Comics	Marvel Comics	5M-10M	33	1	27	79%
Free Movies	Media & Vedio	Movie player	10M-50M	1	0	1	100%
Weather Radar Widget	Weather	Weather alert & forecast	1M-5M	11	0	5	45%
Caller ID & Mobile Locator	Communication	Track phone number location	1M-5M	11	0	11	100%
My Diary	Lifestyle	Write Diaries	500k-1M	29	1	26	86%
Blood Pressure Log - MyDiary	Medical	Blood Pressure Log	100K-500K	5	0	5	100%
Go Fund Me	Social	Personal fundraising platform	1M-5M	41	1	13	29%
Open Camera	Photography	Free camera app	5M-10M	2	0	1	50%
Timesheet - Work Time Tracker	Business	Managing working hours	500K-1M	26	2	13	42%

to obfuscation or proguard. Note that the limitation of applying Aladdin on a closed source app is not due to the complexity of the app, but the recurrence of bytecode obfuscation in apps. In order to make the evaluations comprehensive, we choose one popular app from each category of Google Play except games. We prefer open source apps with the number of downloads exceeding 100K. If no open source apps can meet the requirement, then we choose alternatives that can be successfully analyzed by Aladdin. Table 3 shows all the apps chosen for evaluations. There are 20 apps in total for evaluations[3].

6.2 Coverage

To answer **RQ1**, we perform static analysis to all the apps in our dataset to derive deep links to activities. We manually valid whether each deep link can reach the target activity. The total number of activities, the current number of deep links, and the number of deep links by Aladdin are reported from Column 5 to Column 7 in Table 3. It can be observed that, although the current number of deep links in these apps is very low (the median is 0), Aladdin can derive deep links for all the apps (with median of 9). In particular, all the activities of 5 apps out of the total 20 apps can have deep links after being analyzed by Aladdin, indicating 100% coverage.

However, coverage does not reach 100% in some apps. The main reason is that the intents of some activities encapsulate the objects of app-specific classes, which cannot be resolved by our current solution. We leave this issue for future work.

Considering the improvement of coverage, the median is 52% and the minimum is 29%. Therefore, we can conclude that the static analysis of Aladdin can substantially improve the coverage of deep links of existing apps.

6.3 Accuracy of Dynamic Analysis

We select 4 apps (NPR News, AnkiDroid Flashcards, APUS Booster+, and Wikipedia) in our dataset to which we can apply dynamic analysis of Aladdin. For each app, we randomly choose 2 activities to find fragments. We also manually examine the number of fragments to get the ground truth for comparison. Table 4 shows the analysis results. Here we use recall and precision to measure the accuracy. The recall is the ratio of fragments correctly found by the dynamic analysis over all the expected fragments. The precision is the ratio of fragments correctly found by the dynamic analysis over all the fragments reported by the dynamic analysis. For 5 out of the 8 activities, the precision and recall are both 100%, meaning that the dynamic analysis has found all the fragments as expected.

The current dynamic analysis of fragment extraction does have some limitations. In the activity `Statistics` of *AnkiDroid Flashcards*, the result is smaller than expected. Despite of a 100% precision, the recall is only 56%,

[3]We are continuing to evaluate Aladdin on more apps. Please refer to `http://sei.pku.edu.cn/~mayun11/aladdin/` for more details.

Table 4: Accuracy of dynamic analysis.

App	Activity	Exp.	Result	Recall	Precision
NPR News	SearchActivity	4	4	100%	100%
NPR News	NewsStoryActivity	1	1	100%	100%
AnkiDroid Flashcards	Statistics	9	5	56%	100%
AnkiDroid Flashcards	DeckPicker	11	12	100%	92%
APUS Booster+	HomeActivity	3	3	100%	100%
APUS Booster+	BoostMainActivity	1	4	100%	25%
Wikipedia	SettingsActivity	4	4	100%	100%
Wikipedia	GalleryActivity	5	5	100%	100%

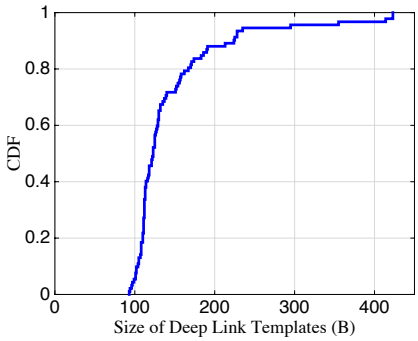


Figure 8: Distribution of the size of the deep link templates.

meaning that developers may have to manually add missing fragments. We find the reason to be that the view which triggers the fragment transition does not have an explicit resource id. To solve this issue in the future, we can use the relative location in the view tree that we build at runtime to identify each view rather than relying on only resource id. On the other hand, in `DeckPicker` of *AnkiDroid Flashcards* and `BoostMainActivity` of *AnkiDroid Flashcards*, the recall is 100% but the precision may not be satisfactory. The reason is the limitation of the structure hash. At runtime, some popping messages and other trivial changes to the original page can result in a total different hash value. As a result, our dynamic analysis should judge the same fragment as a new one. Therefore, taking into account only a single hash value may cause some mistakes in some cases. One possible solution is to record the whole view tree, and design an algorithm to calculate the differences of the view tree after one action. Additionally, only if the difference value reach the threshold can we identify a new fragment.

6.4 Overhead

• **Code Size Increase**. Aladdin packages the source code of the original app logic together with the code of deep-link proxy and deep-link templates when releasing the `.apk`. The core logic of the deep-link proxy is **2,319 lines of Java code** and its size is only 489KB including some third-party libraries. In other words, the size of proxy is rather small compared to that of the original apps. For the size of the deep link templates, we compute the size of the deep link templates for all the 20 apps in our dataset. According to Table 3, there are 295 deep link templates generated by Aladdin in total. Figure 8 shows the distribution of the size. The medium size is 123B, the smallest is 93B, and the largest is 423B. Assume that an app has 1,000 activities, the total code size of templates is no more than 500KB. In other words, Aladdin introduces very tiny code volume compared to the original `.apk` of the app.
• **Runtime Overhead**. Then we evaluate the runtime overhead when requesting a deep link. Such overhead comes from executing the replay engine in the deep-link proxy. We use a `Nexus 5` smartphone model (2.3GHz CPU, 2GB RAM) equipped with Android 6 to launch apps. For each app in the dataset, we first launch the original app on the smartphone and interact with the app for one minute. We record the average CPU usage and memory consumption during the period. Then we launch the app released by Aladdin and we execute 10 deep links to open 10 pages inside the app. We also record the average CPU usage and memory consumption. Evaluation results show that

the Aladdin-packaged app incurs merely no more than 10% of CPU usage and 1MB-3MB memory consumption, which are relatively small compared with the original apps. Therefore, the runtime overhead of apps released by Aladdin is also acceptable.

7 Discussion

Deep links are needed for both informational pages (like tweets in Twitter) and functional pages (like searching places in Google Maps). With deep links, informational pages can be indexed by search engines, and functional pages can be integrated with third-party apps or can provide shortcuts to functionalities for users. Both of the two cases can benefit app developers by introducing more users. For example, messaging apps can provide a deep link to the chat page for sharing news from other apps with friends. Flashlight apps can provide a deep link to its amazing lightening functionality for users to create shortcut without having to configure every time. Therefore, in this paper, we assume that all the pages, referred by activities, are potential locations for generating deep links. But Aladdin enables developers to decide which locations to be equipped with deep links.

Aladdin derives deep-link templates and release deep link schemas for Android apps, and thus provides the infrastructural support of wider usage scenario of deep links. An orthogonal issue is how to retrieve the values of parameters to get a concrete deep link. Indeed, such an issue is dependent on the exact scenarios of how deep links are used. When having the deep links, search engines such as Google and Bing can crawl every page of the app, and thus could get all the possible values for the crawled page. It is straightforward to construct the mapping between the actual values and the schema. App developers could provide additional RESTful APIs to their partners or third-party developers to retrieve the concrete deep links. Since the system can also retrieve all the running information of apps, Android system could leverage the deep-link schema to construct concrete deep links. In this way, users can add a deep link just like "bookmark" and further share it, which is a similar experience on the Web.

Our approach essentially relies on `Intent` to distinguish activity transitions. Different instances of intent could transit to different instances of an activity. We also assume that activity transitions can be re-executed by re-sending the intent. However, in practice, some activity transitions do not instantiate different intents. Instead, they use public objects to pass messages. In such a case, the intent instances of activity transitions are always the same so that our approach may not work. One possible solution is to combine taint analysis to find the data related to activity transitions.

Indeed, importing deep links to apps may have security and privacy issues. Currently, Aladdin relies on the developers to decide which activities to support deep links or not. For simplicity, we assume that all the instances are permitted to be deep "linked". As a result, there can be risks to leak sensitive data and be attacked by hackers. Some preliminary techniques [31, 23] can be combined in Aladdin to alleviate such threats.

Our work in this paper makes the preliminary effort to facilitate the developers who are willing to release deep links. Although the techniques in this paper is for Android apps, the basic principle itself can be customized and applied to other platforms such as iOS and Windows Mobile.

8 Related Work

8.1 Deep link

Deep link [8] is an emerging concept for mobile apps. The idea of deep link for mobile apps originates from the links in the Web. Recently, some great companies, especially search engines, have made many efforts on deep links and proposed their criteria for deep links. Google App Indexing [7] allows people to click from listings in Google's search results into apps on their Android and iOS devices. Bing App Linking [4] associates apps with Bing's search results on Windows devices. Facebook App Links [6] is an open cross platform solution for deep linking to content in mobile apps. However, these state-of-the-art solutions all require the need-to-be-deep-linked apps to have corresponding webpages, and the application is narrowed.

The research community is at the early age of studying deep links and very few efforts have been proposed. Azim et al. [16] designed and implemented uLink, a lightweight approach to generating user-defined deep links.

uLink is implemented as an Android library with which developers can refactor their apps. At runtime, uLink captures intents to pages and actions on each page, and then generates a deep link dynamically, just as bookmarking. Compared to uLink, Aladdin requires smaller developer effort and no intrusive to apps' original code. In addition, Aladdin computes the shortest path to each activity so that we can open a page more quickly than uLink.

8.2 Analysis of Inter-Component Communication

Executing a deep link is highly related to Inter-Component Communication of apps. Paulo et al. [17] presented static analysis for two types of implicit control flow that frequently appear in Android apps: Java reflection and Android intents. Bastani et al.[18] proposed a process for producing apps certified to be free of malicious explicit information flows. Damien et al. [30][29] developed a tool to analysis the intents as well as entry and exist points among android apps. In their following work [28], they show how to overlay a probabilistic model, trained using domain knowledge, on top of static analysis results, in order to triage static analysis results.

8.3 Automated App Testing

Aladdin essentially draws lessons from existing app testing efforts [21][20][32][12][13], and combines the test inputs generation methodology for the dynamic analysis. Google Android development kit provides two testing tools, Monkey [10] and MonkeyRunner [11]. Hu and Neamtiu [24] developed a useful bug finding and tracing tool based on Monkey. Shauvik et al. [22] presented a comparative study of the main existing test input generation techniques and corresponding tools for Android. Ravi et al. [19] presented an app automation tool called Brahmastra to the problem of third-party component integration testing at scale Machiry et al. [27] presented a practical system Dynodroid for generating relevant inputs to mobile apps on the dominant Android platform. Azim et al. [15] presented A3E, an approach and tool that allows substantial Android apps to be explored systematically while running on actual phones.

9 Conclusion

In this paper, we have presented an empirical study of deep links on 20,000 Android apps. Although more and more apps have supported deep links, the coverage of deep links is still very low and implementing deep links requires non-trivial developer manual efforts. To address the issues, we propose Aladdin, a novel approach to automatically release deep links for Android apps. Leveraging the static analysis and dynamic analysis of Android apps, Aladdin can derive deep links to arbitrary locations inside an app. Aladdin integrates a deep-link proxy together with the original source code of the app to realize executing deep links at runtime. So there are no changes to the original business logic and no coding effort from developers. The evaluations on 20 apps demonstrate that the coverage of deep links can be increased by 52% on average while incurring minimal runtime overhead.

References

[1] Android guide. http://developer.android.com/guide/components/index.html.

[2] App links in android 6. https://developer.android.com/training/app-links/index.html.

[3] Baidu app link. http://applink.baidu.com.

[4] Bing app linking. https://msdn.microsoft.com/en-us/library/dn614167.

[5] Deeplinkdispatch. https://github.com/airbnb/DeepLinkDispatch.

[6] Facebook app links. https://developers.facebook.com/docs/applinks.

[7] Google app indexing. https://developers.google.com/app-indexing/.

[8] Mobile deep linking. https://en.wikipedia.org/wiki/Mobile_deep_linking.

[9] Mobile deep linking. http://mobiledeeplinking.org/.

[10] Monkey. http://developer.android.com/tools/help/monkey.html.

[11] Monkeyrunner. http://developer.android.com/tools/help/MonkeyRunner.html.

[12] Ranorex. http://www.ranorex.com/.

[13] Robotium. https://github.com/RobotiumTech/robotium.

[14] Universal links in iOS 9. https://developer.apple.com/library/ios/documentation/General/Conceptual/AppSearch/UniversalLinks.html.

[15] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, volume 48, pages 641–660. ACM, 2013.

[16] T. Azim, O. Riva, and S. Nath. uLink: Enabling user-defined deep linking to app content. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2016*, pages 305–318, 2016.

[17] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 669–679. IEEE, 2015.

[18] O. Bastani, S. Anand, and A. Aiken. Interactively verifying absence of explicit information flows in Android apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 299–315. ACM, 2015.

[19] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium, USENIX Security 2014*, pages 1021–1036, 2014.

[20] N. Boushehrinejadmoradi, V. Ganapathy, S. Nagarakatte, and L. Iftode. Testing cross-platform mobile app development frameworks (t). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 441–451. IEEE, 2015.

[21] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, volume 48, pages 623–640. ACM, 2013.

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

[23] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, pages 1025–1035, 2014.

[24] C. Hu and I. Neamtiu. A GUI bug finding framework for Android applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1490–1491. ACM, 2011.

[25] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng. Characterizing smartphone usage patterns from millions of Android users. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement, IMC 2015*, pages 459–472, 2015.

[26] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, G. Huang, and F. Feng. PRADA: Prioritizing Android devices for apps by mining large-scale usage data. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 3–13, 2016.

[27] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[28] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 469–484, 2016.

[29] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 77–88, 2015.

[30] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium*, pages 543–558, 2013.

[31] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of the 22th USENIX Security Symposium, USENIX Security 2013*, pages 527–542, 2013.

[32] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 506–511, 2015.