# Paladin: Practical Automation of Deep Links Release on Android

Anonymous Author(s)

## ABSTRACT

Compared to the Web where each web page has a global URL for external access, a specific "page" inside a mobile app cannot be easily accessed unless the user performs several steps from the landing page of this app. Recently, the concept of "deep link" is expected to be a promising solution and has been advocated by major service providers to enable targeting and opening a specific page of an app externally with an accessible uniform resource identifier. This paper makes a large-scale empirical study to investigate how deep links are really adopted, over 25,000 Android apps. To our surprise, we find that deep links have quite low coverage, e.g., more than 70% and 90% of the apps do not have deep links on Wandoujia and Google Play, respectively. One underlying reason is the mandatory and non-trivial manual efforts of app developers to support deep links. We then propose the Paladin approach along with its supporting tool to help developers practically automate the release of deep links to access locations inside their apps. Paladin includes a novel cooperative framework by synthesizing the static analysis and the dynamic analysis while minimally engaging developers' inputs and configurations, without requiring any coding efforts or additional deployment efforts. We evaluate Paladin with 579 popular apps and demonstrate its effectiveness and performance.

## KEYWORDS

Deep link; Android apps; Program analysis

**ACM Reference Format:**
Anonymous Author(s). 2018. Paladin: Practical Automation of Deep Links Release on Android. In *Proceedings of The Web Conference (WWW'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

One key factor leading to the great success of the Web is that there are hyperlinks to access web pages and even to specific pieces of "*deep*" web contents. For example, the hyperlink *https://en.wikipedia.org/wiki/World_Wide_Web#History* points to the "*history*" section of the "*WorldWideWeb*" wiki page. Indeed, the hyperlinks play a fundamental role on the Web in various aspects, e.g., enabling users to navigate among web pages and add bookmarks to interested contents, and making search engines capable of crawling the web contents [18].

In the current era of mobile computing, mobile applications (a.k.a., apps) have become the dominant entrance to access the Internet [11, 40]. However, compared to the Web, the support for "*hyperlinks*" is inherently missing in mobile apps so that users have to perform tedious and trivial actions to access a specific in-app content. Other advantages 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) [9]. 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, various major service providers such as Google [7], Facebook [6], Baidu [3], and Microsoft [4], have strongly advocated deep links, and major mobile OS platforms such as iOS [16] and Android [2] have encouraged their developers to release deep links in their apps. Indeed, deep links bring various benefits to current stakeholders in the ecosystem of mobile computing. Mobile users can have better experiences of consuming in-app content by directly navigating to the desirable target app pages. App developers can open their deep links to others who are interested in the content, data, or functionality of their apps, so that they can find potential collaborators to realize the "composition" of apps.

However, it is unclear how deep links have been supported so far in the state of the broad practice. To address such an issue, this paper first conducts an empirical study on popular apps from Wandoujia[1] and uncovers the following findings:

- **An increasing number of deep links with app-version evolution**. Comparing the first version and latest version of the top 200 apps in Wandoujia, the percentage of apps that support deep links increases from 30% to 80%;
- **Low coverage of deep links of current apps**. Although it is observed that the number of apps supporting deep links keeps increasing, the current coverage of deep links is rather low: for apps with deep-link support, only 4% of activities actually have deep links;
- **Non-trivial efforts from developer.** We make in-depth study of deep-link support at source-code level. Based on our study of open-source Android apps from GitHub, developers need to manually modify 45–411 lines of code to implement deep links to one activity.

In order to release deep links with minimal developer efforts, in this paper, we propose ***Paladin***, a novel approach that helps developers **p**ractically **a**utomate the re**l**ease of **A**ndroid app's **d**eep l**in**ks 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 an Android app, Paladin first uses static analysis to find how to reach activities (each of which is the basic UI component of Android apps) inside the app most efficiently from the entrance of the app. After developers select activities of which the dynamic locations need to be deep linked, Paladin then performs dynamic analysis to find how to reach fragments (each of which is a part of a UI component) inside each activity. Finally, Paladin synthesizes the analysis results and generates the templates that record the scripts of how to reach a location inside the app. Paladin provides a deep-link proxy integrated with the app code 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 Paladin on 579 popular Android apps. The evaluation

*WWW'18, April 2018, Lyon, France*
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Wandoujia now owns over 3.5 million users and 2 million Android apps. See its website via http://www.wandoujia.com.

results show that deep links released by Paladin can cover a large portion of an app, and the runtime performance is desirable.

To the best of our knowledge, Paladin is the first work to automate the release of deep links of Android apps without any intrusion of their normal functionality, and thus establishes the foundation of "*web-like*" user experiences for mobile apps. More specifically, this paper makes the following major contributions.

- We conduct an extensive empirical study of current deep links based on 25,000 popular Android apps, uncovering the current status of deep links and the developer efforts to implement deep links.
- We propose an approach to helping developers practically automate the release of deep links based on a cooperative framework, with developers' only very minimal configuration efforts and no interference of the normal functionalities of an app.
- We evaluate the effectiveness of our approach on popular Android apps.

# 2 BACKGROUND

In this section, we present some background knowledge of Android apps and deep links.

## 2.1 A Conceptual Analogy

An Android app, identified by its package name, usually consists of multiple `Activities` that are loosely bound to each other. An activity is a component that provides a user interface that users can interact with, such as dialing phones, watching videos, or reading news. Each activity 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.

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 package name 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. The main activity is just like the home page of a website.

An activity has several view components to display its user interface, such as `TextView, ButtonView, ListView`. View components 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, which 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.

Transitions between activities are executed 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. Note that intents can be used to transit between activities from both the same apps and two different apps. There are two types of intents: (1) *explicit* intents, which specify the target activity by its class name; (2) *implicit* intents, which declare `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 Android apps and the Web.**

| Concepts of Android Apps | Concepts of Web |
|---|---|
| app | website |
| package name | domain |
| activity | web page |
| main activity | home page |
| view component | web element |
| fragment | frame |

## 2.2 Deep Links

The idea of deep links for mobile apps originates from the prevalence of hyperlinks on the Web. Every single web page has a globally unique identifier, namely URL. In this way, web pages are accessible directly from anywhere by means of URLs. 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.

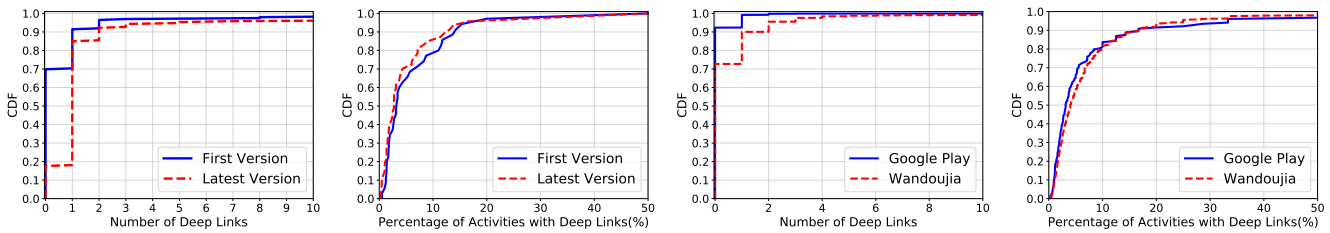
Compared to web pages, the mechanism of hyperlinks is historically missing for mobile apps. To solve the problem, deep links are proposed to enable directly opening a specific page inside an app from the outside of this app with a uniform resource identifier (URI). The biggest benefit of deep links is not limited in enabling users to directly navigate to specific locations of an app, 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. Below are some usage scenarios of deep links.

- **In-App Search**. In-app search [7] enables mobile users to search contents inside apps and enter directly into the app page containing the information from search results.
- **Bookmarking**. Mobile users can create bookmarks to the information or functionalities inside apps for later use [18].
- **Content Sharing**. With deep links, mobile users can share pages from one app to friends in social networking apps [6].
- **App Mashup**. Other than simple content sharing, deep links can further act as the support for realizing "*app mashup*" [34] to integrate services from different apps, like IFTTT [8].

# 3 AN EMPIRICAL STUDY

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, according to Android's developer guide [1], 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




(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 1: The trend and status of deep links among Android apps.**

the `android.intent.category.BROWSABLE` category. We denote these intent filters as deep-link related. If an activity has deep-link related intent filters, then we say the activity is registered with deep links. Therefore, we can take the number of activities registered with deep links as an indicator to estimate the number of deep links for an Android app. We should mention that activities without deep-link related intent filters may still register deep links because developers can register a single activity with deep links and forward the link request to other activities. As a result, our estimation of deep links may be below the actual number. However, such a case usually exists in apps that have only one activity registered with deep links. So our results are still able to reveal the status of deep links in practice.

## 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. We choose top 200 apps (as of Jan. 2017) ranked by Wandoujia, a leading Android app store in China [30, 33]. To make comparison, we manually download each app's first version that could be publicly found on the Internet and its latest version published in Wandoujia as of Jan. 2017. We compare the number of deep links in the two versions of each app. Figure 1(a) shows the distribution of the number of deep links among all the apps in each version. Generally, it is observed that when first released, only 30% of apps have deep links. In contrast, more than 80% of these apps have supported deep links in their latest versions. More specifically, the maximum number of deep links is 35 in the first version and it increases to 81 in the latest version. Such a change indicates that 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, the percentage of activities that have deep links in deep-link supported apps is still rather low. We compute the ratio of activities with deep links to the total number of activities. As shown in Figure 1(b). It is surprising to find that the percentage of activities with deep links becomes smaller from the first to the latest version. For the first version, there are more than 20% of apps of which the percentage of activities with deep links is above 10%, but the number of apps decreases to 15% for the latest version. The reason is that developers add more activities when upgrading their apps but they release deep links to only fixed activities.

Aiming to expand the investigation to a wide scope, we study the latest version of 20,000 popular apps on Wandoujia, and 5,000 popular apps on Google Play as of Jan. 2017. Figure 1(c) shows the distribution of the number of deep links among apps in the two app markets. Similar to the preceding results, more than 70% and 90% of the apps do not have deep links on Wandoujia and Google Play, respectively. Such a result indicates that deep links are not well supported in a large scope of current Android apps, especially the international apps on Google Play.

**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 |

Considering the percentage of activities with deep links, Figure 1(d) shows that the median percentage is just about 4%, implying that a very small fraction of the locations inside apps can be actually accessed via deep links. About only 10% of apps have more than 20% of activities with deep links. There is no significant difference between the distribution on Wandoujia and Google Play, meaning that the low coverage of deep links is common for the Android ecosystem.

In summary, the preceding empirical study demonstrates the low coverage of deep links in current Android apps. Such a result is a bit out of our expectation, since deep links are widely encouraged in industry to facilitate in-app content access. There are four possible reasons leading to the low coverage of deep links. First, as deep link is a relatively new concept, it may take some time to be adopted by Android developers. Second, due to commercial or security considerations, developers may not be willing to expose their apps to third parties through deep links. Third, the developers do not have clear motivation to determine which part of their apps needs to be exposed by deep links. Fourth, as we show later, implementing deep links requires non-trivial developer efforts so that developers may not be proactive to introduce deep links in their apps. However, deep link is still promising in the mobile ecosystem given the strong advocation by major Internet companies as well as the potential revenue brought by opening data and cooperating with other apps.

## 3.3 Developer Efforts

Indeed, supporting deep links requires the developers to write code and implement the processing logics, not just by declaring the intent filter in the AndroidManifest.xml file. Although there have already

been some SDKs for deep links [5, 10], implementing deep links exactly requires the modifications or even refactoring of the original implementation of the apps. We then study the actual developer efforts 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 all the results, we find 8 projects that actually add deep links in their code-commit history. We carefully check the code changes in the commit related to deep links. Table 2 shows the LoC (lines of code) of changes in each project when adding deep links to just *one* activity in the corresponding code commit. The changes include the addition, modification, and deletion of the code. We can observe that the least number of the LoC change is 45 and the biggest can reach 411. Take the app `SafetyPongAndroid`[2] as an example. 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. Several objects that are previously initialized in other activities need to be initialized in the activity to be deep linked. 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 one of the potential reasons why deep links are of low coverage.

# 4 PALADIN: 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 make deep links practically and broadly supported by existing apps, we propose the ***Paladin*** approach to automate the release of deep links for Android apps. The design goals of Paladin are four aspects:

**Maximal deep-link coverage**. Paladin should release deep links to as many activities as possible for developers to choose the desirable locations to actually expose deep links.

**Supporting fine-grained deep links**. Current deep links point to only activities. However, it becomes popular that activities have multiple fragments for different features. Paladin should release deep links not only to activities, but also to fragments for better user experience.

**Minimal developer efforts**. Paladin should automate the process of releasing deep links and require few or zero coding efforts.

**Minimal runtime overhead**. The performance of requesting deep links generated by Paladin should be efficient enough to ensure user experience.

Figure 2 shows the overview of Paladin. Paladin achieves the preceding goals via designing a cooperative framework of program analysis and a proxy architecture of deep-link execution. More specifically, Paladin's cooperative framework combines static program analysis and dynamic program analysis while minimally engaging developers to provide inputs to obtain apps' execution paths that can be re-executed afterwards. Each path represents a deep link to a specific location inside apps. Leveraging the Android testing framework, Paladin's proxy architecture can enable deep-link execution without having to modify the app code. We next present the details of our approach.

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

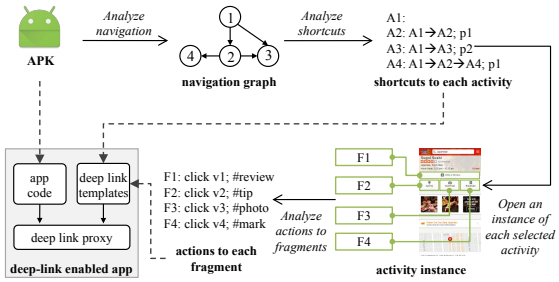



**Figure 2: Approach overview.**

## 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, instead of a single intent to an activity, Paladin 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 flow 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 $t(\mathcal{L})$ is triggered by an intent, where $\mathcal{L}$ 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 $\mathcal{L}$ to abstract an intent. Two intents are equivalent if and only if the label sets are completely the same. Note that from 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 $t(\mathcal{L})$; $r$ 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 analyze the paths to each activity.

Definition 3 (Path). *A path to an activity $\mathcal{P}_a$ 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 $\mathcal{P}_a$ 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. 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 the system level. Therefore, for each activity, we compute the shortest path, denoted as the shortcut, and the combination of labels in the activity transitions of the path constitutes the label set of the shortcut.

Definition 4 (Shortcut). *A Shortcut $\mathcal{T}(\mathcal{L})$ of an activity $a$ is the shortest path $\mathcal{P}_a = \{t_i\}$, and $\mathcal{L} = \cup \mathcal{L}t_i$.*

## 4.2 Deriving Deep Links to Fragments

As shown in Section 2, there are different fragments in an activity for serving 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 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 class name is the identifier of different activities, fragments usually do not have explicit identifiers. 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 1 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 $r$'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.

*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.

**ALGORITHM 1:** Computing structure hash of view tree.

```
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)
```

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 is 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. For each view in the current fragment, we try to click it 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 the previous condition. 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. Therefore, we can add it into the edge set if it is a new fragment. The dynamic analysis is similar to the web crawlers that need to traverse every web page, except that Android provides only 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.

After finishing the traverse search, we can get the fragment transition graph and a list of fragments. To get the action path towards a certain fragment, we simply combine all the edges from the start vertex to the fragment.

## 4.3 Releasing Deep-Link Supported Apps

After computing the shortcuts to activities and action paths to fragments, the next step is to create the target APK file that supports processing deep links at runtime. Note that developers may want to create deep links to only some locations of their apps. Therefore, Paladin allows developers to configure the locations to be deep linked, including the activities and fragments inside activities. Then for each selected location, Paladin generates a deep-link template by combining the shortcuts of the corresponding activity and the action path of the corresponding fragment.

Next, Paladin 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

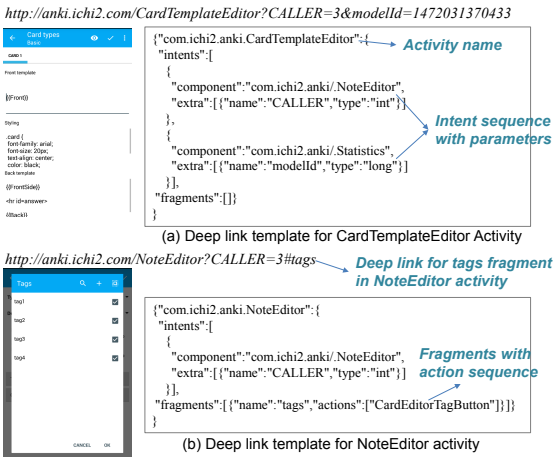



**Figure 3: Example of deep link templates.**

abstract URI. We use the reverse string 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 `#`. For each abstract URI, we can generate an intent filter with the URI's schema in the AndroidManifest.xml file to handle the corresponding deep links.

Figure 3 shows two deep link templates of the `Anki` app released by Paladin. 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 Paladin can populate the proper intents to transfer to the target activity. `NoteEditor` has a fragment naming "*tags*". The action to the fragment is clicking the view whose resource id is `CardEditorTagButton`. Therefore, to reach the tags fragment, not only should the value of intents be assigned (*CALLER* = 3), but the fragment should be specified as well (#*tags*).

Figure 4 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 *Execution 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 *App Code* for default execution.

When an deep link is executed, the corresponding deep-link template is instantiated with concrete values to create a execution script. Then the *Execution Engine* communicates with the original *App Code* and instructs the app to transit through activities and perform actions on views according to the script. For example, in Figure 3(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 Execution Engine first issues the intent with the parameter `CALLER` is 3. Then the Execution Engine performs a click on the view whose resource

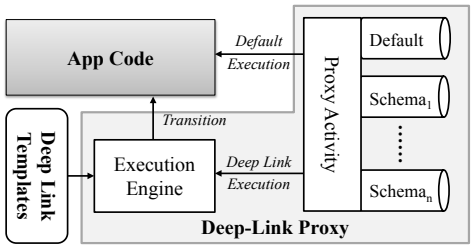



**Figure 4: Structure of the app with deep link enabled.**

id is `CardEditorTagButton`. Finally, the user can reach the target location.

## 5 IMPLEMENTATION

To construct the *Navigation Graph*, we first use the `IC3` tool [37] to extract all the activities and intents from the APK file of an Android app. Then we apply the `PRIMO` tool [36] 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 connect 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 the 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 file with deep link enabled, the *Execution Engine* is actually implemented as a test case of the instrumentation test to execute the deep-link templates with parameter values. 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 Execution Engine. The Execution Engine uses the instrumentation object provided by the framework to send intents in the app process, reaching the target activity. Then it sends action events to finally reach the target fragment.

## 6 EVALUATIONS

In this section, we evaluate Paladin from four aspects, i.e., the coverage improvement of deep links to activities, the effectiveness of releasing deep links to fragments, the performance of executing deep links, and the overhead brought by Paladin.

### 6.1 Deep Link Coverage of Activities

We first investigate to what extent Paladin can help release deep links to activities. We collect a representative dataset by choosing top 50 apps from each of the 14 categories (without Game) in Wandoujia. Then we perform Paladin's static analysis to all the apps in our dataset, and calculate the percentage of activities that can be released with deep links after applying Paladin. Due to the bytecode obfuscation and proguard, some of the apps fail to be analyzed by

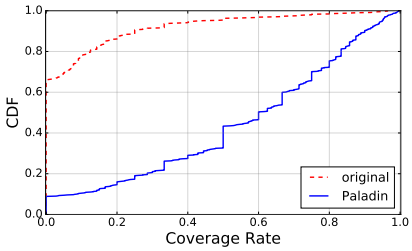



**Figure 5: Distribution of the percentage of deep-linked activities before and after applying Paladin.**

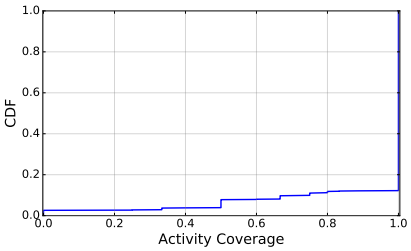



**Figure 6: Distribution of the percentage of reached activities for which we gather at least one deep link.**

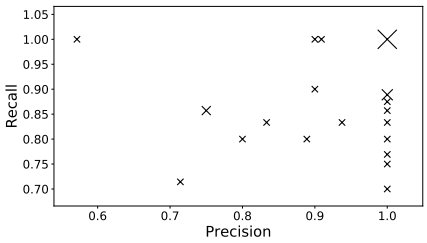



**Figure 7: The recall and precision of identifying fragments by Paladin's dynamic analysis.**

Paladin. Such failures are due to the limitation of the static analysis tool that we use to implement Paladin, not the Paladin approach itself. We finally get results from 579 apps. Figure 5 shows the distribution of the percentage of "deep-linked" activities before and after applying Paladin. Apparently, the current coverage of deep links in these apps is very low (the median coverage is 0). In contrast, applying Paladin can derive deep links for more than 90% of these apps (the median coverage reaches 60%). In particular, 55 apps (about 9%) have more than 90% of deep linked activities after Paladin is applied. Such an observation indicates that Paladin can effectively release deep links for most activities, thus improving the coverage of deep links. Note that the coverage does not reach 100% after applying Paladin because Paladin handles the intents whose parameters are all basic types, but intents to some activities actually have complex objects of app-specific classes.

To evaluate whether the Paladin-released deep links can correctly access the corresponding activities in an app, we further run every single app in the dataset to collect a set of concrete deep links. For each app, we use the popular Monkey tool [12] to randomly generate 1,000 events every 60 ms, and retrieve the intents from ADB when activities are switched. We record all the reached activities as well as the corresponding intents. Then we filter out those activities where deep links have been released by Paladin, extract parameter values from the intents, and assign the values to the corresponding deep-link schema to instantiate concrete deep links. Then we request each concrete deep link to check whether the same activity instance as the recorded one can be reached. If the requested activity can be accessed, we can confirm that the derived deep link is correct. Results show that *all the deep links can be correctly executed to the target activity instance*, indicating that the correctness of the Paladin-released deep links is well guaranteed.

Additionally, the collected activities could represent the commonly used functionalities in an app by the users. Figure 6 shows the distribution of the percentage of the reached activities where we gather at least one deep link among all the apps. It is observed that for more than 90% of apps, all the reached activities are released with deep links by Paladin. Such a result implies that Paladin can release deep links to most of the commonly used activities in an app.

## 6.2 Deep Link Effectiveness of Fragments

We then evaluate how Paladin's dynamic analysis can effectively release deep links to fragments. From the 579 apps in Section 6.1, we select 30 apps whose main activity has at least 5 fragments. Then we apply Paladin's dynamic analysis to identify fragments for the main activity in each app. The number of fragments found by the dynamic analysis is denoted as $N$. We also manually explore the main activity in each app to examine the number of desired fragments (denoted as $D$) as the ground truth for comparison. Next, we manually compare the screenshots between the fragments identified by Paladin and the manually examined fragments, and the number of the same fragments in both sets is denoted as $C$. Therefore, we can use the *recall* (which is $C/D$) and *precision* (which is $C/N$) to measure the effectiveness of Paladin's dynamic analysis.

Figure 7 shows the scatter diagram of the recall and precision for the 30 apps. Each cross in the diagram represents the apps with the same recall and precision value. We can see that for 9 out of the 30 apps, both the precision and recall can reach 100% (represented by the biggest cross in the top right corner), indicating that our dynamic analysis has just found all the fragments as desired and no redundant fragments are identified.

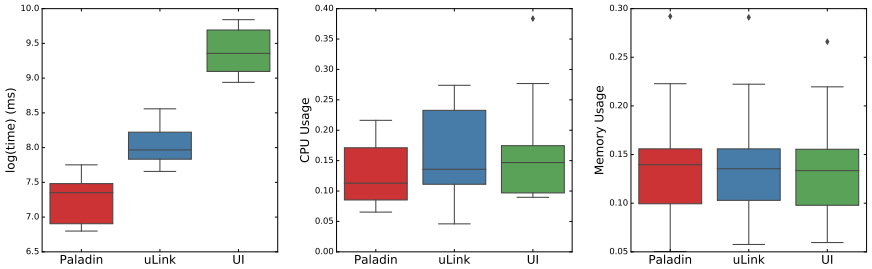
For 10 apps, the precision reaches 100% but the recall does not, indicating that all the identified fragments are desired but some desired fragments are omitted. As a result, developers have to manually configure those missing fragments. The reason is that the view triggering the fragment transition does not have an explicit resource ID and thus cannot be found by the dynamic analysis. To address this issue, we can use the relative location in the view tree that we build at runtime to identify every single view rather than relying on only the resource ID.

For 3 apps, the recall reaches 100% but the precision does not, indicating that Paladin has identified all the desired fragments but there exist some redundant results, i.e., more than two identified fragments are actually mapped to one desired fragment. The reason is the limitation of the structure hash calculated by Algorithm 1. At runtime, some popup messages and other trivial changes to the original view tree can result in a total different hash value. As a result, the dynamic analysis process treats the same fragment as a new one. Therefore, taking into account only a single hash value can cause some mistakes. One possible solution is to record the whole view tree, and design an efficient algorithm to calculate the differences of the view tree after the action is performed.

For other apps, both recall and precision do not reach 100%. However, the lowest recall and precision are about 70% and 55%, respectively. Such results indicate that more than 70% of the desired fragments can be identified and no more than half of the identified results are redundant. Therefore, Paladin's dynamic analysis is effective enough for developers to use in practice.

## 6.3 Performance of Executing Deep Links

We compare the performance of executing deep links released by Paladin with two alternative solutions: one is uLink [18], which




(a) Execution time (b) CPU usage (c) Memory usage

**Figure 8: Performance of executing deep links.**

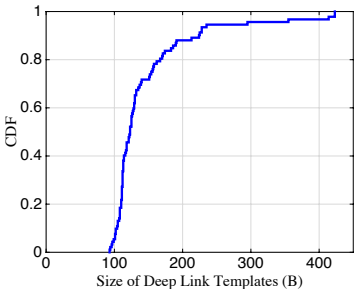



**Figure 9: Distribution of the size of the deep-link templates.**

enables user-defined deep links to Android apps; the other is UI operation, which represents the case where users manually interact with the app to reach the target location. uLink is the only work on deep-link generation that has been published in a peer-reviewed venue. We discuss the difference between Paladin and uLink later in Section 8. Here we just mention one significant difference: for the deep links to activities, Paladin computes the shortest intent path while uLink simply replays all the intents to the target activity because uLink adopts a program-by-demonstration philosophy to extract deep links when users are using the app. For the case of UI operation, we use it as a baseline to investigate the benefits brought by deep links.

Based on the concrete deep links collected (Section 6.1), we select 20 activities from 20 apps. Each activity can be launched via shortcuts, and the length of the original intent path is more than three. We use a `Nexus 5` smartphone (2.3GHz CPU, 2GB RAM) equipped with Android 5.1 to deploy the three alternative solutions. For Paladin, we just deploy the output APK of each app and execute the corresponding deep link to each selected activity. For uLink, since it is not open source, we just remove the computation of shortest path from the implementation of Paladin to simulate the behaviors of uLink. For UI operation, we re-send the captured UI events. To simulate the real user behaviors, after sending one event, we ensure that the screen is fully rendered before we send the successive event. We record the time spent on executing the deep link for each solution. During the execution, we also record the CPU and memory usage information via ADB. We repeat each case three times.

Figure 8 shows the results. Both Paladin and uLink save significant amount of time to reach the target activity compared with UI operation, while the CPU and memory usages are not significantly different. Such a result implies that deep links can efficiently reach deeper locations inside apps, and our implementation of Paladin does not cause noticeable overhead. The execution time of Paladin is two times faster than uLink, demonstrating the benefits of shortcuts. In addition, Paladin's CPU usage is smaller than uLink's while the memory usage is almost the same. The reason may be that Paladin passes a smaller number of activities before reaching the target than uLink does, requiring more CPU resources to execute the transition activity's logic. In summary, Paladin performs better than uLink in terms of both execution time and resource consumption.

## 6.4 Overhead of Paladin

Paladin packages the app code of the original app together with the code of deep-link proxy and deep-link templates when releasing the APK file. The core logic of the deep-link proxy is **2,319 lines of Java code** and its size is only 489 KB including some third-party libraries. In other words, the proxy's size 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 579 apps in our dataset from Section 6.1. There are 7,375 deep-link templates generated by Paladin in total. Figure 9 shows the distribution of the template size among all the templates. The medium size is 123 B, the smallest is 93B, and the largest is 423 B. For the 25,000 most popular apps in Section 3.2, we find that the median number of activities in an app is 160, so the total code size of templates on average is no more than 70 KB. Overall, Paladin introduces very tiny volume compared to the original app.

# 7 DISCUSSION

As the first deep-link automation effort to date, we realize that there are some issues worth discussing to improve the real-world applicability of Paladin.

• **Access to arbitrary locations.** Due to the complexity of mobile apps, there are some special cases which Paladin have not handled so that not all the locations in the app can be released with deep links by Paladin. First, some activities are launched by intents with complex object messages which cannot be constructed externally so that Paladin cannot handle such a case. Second, some activities can be launched only in specific states of the app, e.g., after logging in a account. Otherwise, there may be execution errors. So far, Paladin has not considered such external dependencies of activities. But most apps have implemented complementary logics when the state is not expected. For example, when a user enters an activity that requires a user account but she does not log in, the app may ask the user to log in first and afterwards return back to the previous activity. Therefore, this limitation is not very significant. Third, there may be pop-up views that has to be closed before performing actions on the activity. Currently, Paladin does not consider such a condition so that the fragment may not be correctly reached. In the future work, we plan to enhance Paladin to make it more robust to complex conditions of app executions.

• **Developer-desirable release**. The design goal of Paladin is to help explore all possible locations that can be equipped with deep links. In practice, developers can choose whether the Paladin-generated deep links are required before releasing their apps. Indeed, just like the web pages, not every location inside an app mandatory needs a deep link. Nevertheless, Paladin indeed facilitates developers to more efficiently release their desirable deep links with very little manual efforts. As analyzed previously, releasing

deep links can make their apps more enrich user interactions and embrace more potential collaborators and revenues.

• **Instantialization of deep links.** Paladin essentially derives deep-link templates and releases 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 execute a concrete deep link. Indeed, such an issue is dependent on the exact context where deep links are used. For example, search engines such as Google and Bing can crawl every single page of the app, and thus can get all the possible values for the crawled page.

• **Security and privacy.** Indeed, importing deep links to apps may lead to security and privacy issues. Currently, Paladin 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". However, there might be some potential risks to be attacked by hackers via deep links [32]. To keep our contribution clean, addressing the threats of deep links is out of the scope of this paper. In our future work, we plan to leverage the preliminary WHYPER technique [39] to figure out the possible security and privacy issues.

• **Support of out-of-date content**. Similar to the web hyperlinks, it is argued that the deep links can be out of date and unavailable if the app-content providers remove the content which the deep link refer to [18]. As analyzed previously, Paladin generates deep-link template for the location that the developers desire to be "linked", the initialization of deep links are made at runtime. Hence, Paladin-generated deep links inherently are not affected with respect to the updated or removed content. When the developers decide to add, update, or remove the support of deep links for a location inside their latest released app versions, they just have to apply Paladin to the new-version app and configure the desirable deep links.

• **Generalizability.** Paladin 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 that are implemented in Java, the idea and basic principle itself can be extended and applied to other platforms such as iOS and Windows Mobile. Indeed, the static/dynamic analysis techniques over these platforms shall be different.

## 8 RELATED WORK

In this section, we summarize the related work.

• **Deep Link**. Deep link [9] is an emerging concept for mobile apps. Recently, some major 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. [18] 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, Paladin requires smaller developer efforts and no intrusion to apps' original code. Besides, Paladin computes the shortest path to each activity so as to open a page more quickly than uLink, as shown in Section 6.3. Other possible solutions to implement deep links are to leverage the record-and-replay techniques on mobile devices [26, 29]. However, these tools are too heavy-weight [25] and require either a rooted phone or changes to the mobile OS. So Paladin provides a developer tool to refactor the apps for deep links, achieving both non-intrusion and lightweight execution.

• **Analysis of Inter-Component Communication**. Executing a deep link is highly related to Inter-Component Communication (ICC) of apps. Paulo et al. [19] presented static analysis for two types of implicit control flow that frequently appear in Android apps: Java reflection and Android intents. Bastani et al.[20] proposed a process for producing apps certified to be free of malicious explicit information flows. Li et al. [31] proposed IccTA to improve the precision of the ICC analysis by propagating context information between components. Damien et al. [38][37] developed a tool to analyze the intents as well as entry and exist points among Android apps. In their more recent work [36], they show how to overlay a probabilistic model, trained using domain knowledge, on top of static analysis results to triage static analysis results. Xu et al. [41] analyzes the collusion behaviors among Android apps.

• **Automated App Testing**. Paladin essentially draws lessons from existing app testing efforts [23][22][42][14][15], and combines the test generation methodology for the dynamic analysis. The Google Android development kit provides two testing tools, Monkey [12] and MonkeyRunner [13]. Hu and Neamtiu [28] developed a useful bug finding and tracing tool based on Monkey. Shauvik et al. [24] presented a comparative study of the main existing test generation techniques and corresponding tools for Android. Ravi et al. [21] presented an app automation tool called Brahmastra to the problem of third-party component integration testing at scale. Machiry et al. [35] presented a practical system Dynodroid for generating relevant inputs to mobile apps on the dominant Android platform. Azim et al. [17] presented A3E, an approach and tool that allows substantial Android apps to be explored systematically while running on actual phones. Hao et al. [27] designed PUMA, a programmable UI automation framework for conducting dynamic analyses of mobile apps at scale. Different from these tools and systems, the goal of Paladin's dynamic analysis is to identify fragments or sub-screens of activities that are internal states of apps. So we design UI-tree-based fragment identification and fragment transition graph to address such an issue.

## 9 CONCLUSION

In this paper, we have presented an empirical study of deep links on 25,000 Android apps and proposed the Paladin approach to help developers automatically release deep links. The evaluations on 579 apps have demonstrated that the coverage of deep links can be increased by 60% on average while incurring minimal developer efforts. Some ongoing efforts are making Paladin more practical. First, we are enhancing the static analysis of activities to resolve the objects of app-specific classes encapsulated in the intents to further improve the coverage. Second, we are optimizing the algorithm of dynamic analysis to improve the precision and recall of the fragment coverage. Finally, we plan to apply Paladin to more apps in order to get feedback from app developers for further evaluation.

# 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] Ifttt. https://ifttt.com/.
[9] Mobile deep linking. https://en.wikipedia.org/wiki/Mobile_deep_linking.
[10] Mobile deep linking. http://mobiledeeplinking.org/.
[11] Mobile internet use passes desktop. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds.
[12] Monkey. http://developer.android.com/tools/help/monkey.html.
[13] Monkeyrunner. http://developer.android.com/tools/help/MonkeyRunner.html.
[14] Ranorex. http://www.ranorex.com/.
[15] Robotium. https://github.com/RobotiumTech/robotium.
[16] Universal links in iOS 9. https://developer.apple.com/library/ios/documentation/General/Conceptual/AppSearch/UniversalLinks.html.
[17] 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.
[18] 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.
[19] 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 *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 669–679, 2015.
[20] 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, 2015.
[21] 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 *Proceedings of the 23rd USENIX Security Symposium, USENIX Security 2014*, pages 1021–1036, 2014.
[22] N. Boushehrinejadmoradi, V. Ganapathy, S. Nagarakatte, and L. Iftode. Testing cross-platform mobile app development frameworks. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 441–451, 2015.
[23] 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.
[24] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for Android: Are we there yet? In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 429–440, 2015.
[25] J. Flinn and Z. M. Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications, HotMobile 2011*, pages 84–89, 2011.
[26] L. Gomez, I. Neamtiu, T. Azim, and T. D. Millstein. RERAN: timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering, ICSE 2013*, pages 72–81, 2013.
[27] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2014*, pages 204–217, 2014.
[28] 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, 2011.
[29] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pages 349–366, 2015.
[30] 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.
[31] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering, ICSE 2015*, pages 280–291, 2015.
[32] F. Liu, C. Wang, A. Pico, D. Yao, and G. Wang. Measuring the insecurity of mobile deep links of android. In *Proceedings of the 26th USENIX Security Symposium, USENIX Security 2017*, pages 953–969, 2017.
[33] 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.
[34] Y. Ma, X. Liu, M. Yu, Y. Liu, Q. Mei, and F. Feng. Mash droid: An approach to mobile-oriented dynamic services discovery and composition by in-app search. In *Proceedings of 2015 IEEE International Conference on Web Services, ICWS 2015*, pages 725–730, 2015.
[35] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.
[36] 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.
[37] 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 IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 77–88, 2015.
[38] 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.
[39] 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.
[40] H. Wang, Z. Liu, Y. Guo, X. Chen, M. Zhang, G. Xu, and J. Hong. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017*, pages 163–172, 2017.
[41] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu. Appholmes: Detecting and characterizing app collusion among third-party android markets. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017*, pages 143–152, 2017.
[42] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 506–511, 2015.