uLink: Enabling User-Defined Deep Linking to App Content

Tanzirul Azim*
University of California, Riverside

Oriana Riva
Microsoft Research

Suman Nath
Microsoft Research

ABSTRACT

Web deep links are instrumental to many fundamental user experiences such as navigating to one web page from another, bookmarking a page, or sharing it with others. Such experiences are not possible with individual pages inside mobile apps, since historically mobile apps did not have links equivalent to web deep links. Mobile deep links, introduced in recent years, still lack many important properties of web deep links. Unlike web links, mobile deep links need significant developer effort, cover a small number of predefined pages, and are defined statically to navigate to a page for a given link, but not to dynamically generate a link for a given page. We propose uLink, a novel deep linking mechanism that addresses these problems. uLink is implemented as an application library, which transparently tracks data- and UI-event-dependencies of app pages, and encodes the information in links to the pages; when a link is invoked, the information is utilized to recreate the target page quickly and accurately. uLink also employs techniques, based on static and dynamic analysis of the app, that can provide feedback to users about whether a link may break in the future due to, e.g., modifications of external resources such as a file the link depends on. We have implemented uLink on Android. Our evaluation with 34 (of 1000 most downloaded) Android apps shows that compared to existing mobile deep links, uLink requires minimal developer effort, achieves significantly higher coverage, and can provide accurate user feedback on a broken link.

1. INTRODUCTION

In the web, deep linking refers to the use of hyperlinks to a specific piece of web content (e.g., http://ulink.com/code/) on a website (e.g., http://ulink.com). Web deep links are instrumental to many fundamental user experiences: navigating to one web page from another, bookmarking it, and sharing it with others. They have also been crucial for many important services; for example, search engines use deep links to crawl web pages and to map search results to appropriate landing pages. Historically, mobile apps did not have any equivalent deep links, making the aforementioned tasks impossible for individual pages within the apps. As VentureBeat rightly put, "Imagine a web without URLs. That's what the mobile app world looks like now [July, 2014]" [42].

*Work done while at Microsoft Research.

MobiSys'16, June 25–30, 2016, Singapore.
© 2016 ACM. ISBN 978-1-4503-4269-8/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2906388.2906416

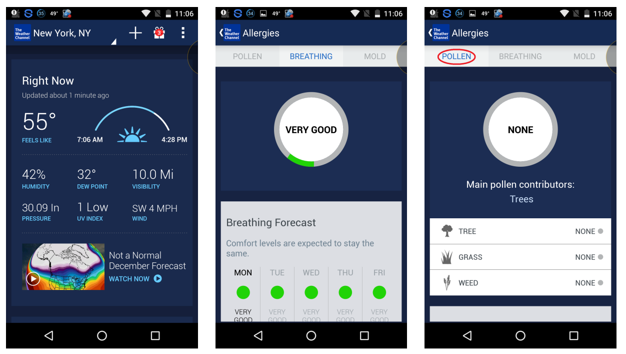
To address this, mobile deep links have been introduced in recent years [6,7,10,39]. Mobile deep links are URIs that point to specific locations within apps. A mobile deep link can launch an already installed app on a user's mobile device (similar to loading the home page of a website) or it can directly open a specific location within the app (similar to deep linking to an arbitrary web page). For example, the URI fandango://thelegomovie_159272/movieoverview directly navigates to the page with the details of the "The Lego Movie" in the Fandango app. Today, all major mobile platforms, including Android, iOS, and Windows, support mobile deep links.

Even though mobile deep linking is an important first step towards randomly accessing any arbitrary location within an app, it lacks many useful properties of web deep linking. First, unlike web deep links, mobile deep links require *nontrivial developer effort*–several lines of codes per deep link—resulting in a low adoption rate even within the top apps [40]. Second, unlike its web counterpart, mobile deep links have *poor coverage*—a small number of locations within an app, predefined by the developer, are directly accessible via deep links (details in §2.3). Finally, today's deep links are *defined statically* by developers to facilitate navigation to a target page given its link; the dual process of dynamically determining the link for a given page is not possible even if a deep link to that page exists.

In this paper we propose *uLink*, a lightweight approach that addresses the above problems. uLink is compatible with existing mobile deep links (i.e., the underlying mobile OS handles them in the same way); but it requires minimal developer effort, it supports dynamic link creation, and it achieves significantly higher coverage than existing mobile deep links. All this enables many novel user experiences that so far existed only in the web world.

One key challenge uLink addresses is improving coverage—creating links to any app location (referred to as *app view* or *view* hereafter), including to the ones that depend on previous views or on user interactions. uLink uses two key mechanisms. The first mechanism is *shortcut*. uLink continuously monitors for explicit data dependency between successive runtime views in an app. In Figure 1, view (a) launches view (b), by providing the location "New York, NY" selected by the user in (a). In some cases, e.g., if (a) and (b) are separate Android activities (i.e., pages), uLink can transparently capture the data transferred from (a) to (b) and encode it in the link to (b). This allows uLink to quickly invoke the link to go to (b), without first going to (a). More importantly, it improves coverage to views that depend on data from previous views (location in this example).

Shortcuts do not cover all app views. The view shown in Figure 1(c) is created when the user taps on the "POLLEN" tab, and



(a) An independent view (b) A view dependent on a previous view (c) A view dependent on a UI action (tap on "POLLEN" tab)

Figure 1. Examples of views uLink can support. Mobile deep links can support only (a).

there is no explicit data transfer between (b) and (c) for uLink to capture—both views are within the same Android activity. To create links to such views, uLink uses a limited form of record and replay. uLink continuously records UI actions in the current view, and encodes them in the link (we call this a *shortcut-and-replay link*). When the link is invoked, uLink first directly navigates to the most recent shortcut-reachable view (e.g., (b) in Figure 1), and then replays the UI actions to navigate to the target view.

uLink's record and replay mechanism is different from existing record and replay techniques that have been successfully used to repeatedly navigate to arbitrary locations of an application for desktop, server [14, 21, 31, 33, 36, 44], web [29, 34], and mobile platforms [19, 23]. Compared to traditional record and replay systems, uLink's record and replay is (1) lightweight and universally deployable—this is because it records and replays only UI events, which, as we show later, is often sufficient to recreate a target view with high fidelity; (2) fast—this is because uLink does not replay a whole session; rather it replays UI events only after reaching the most recent shortcut-reachable view; and (3) user-friendly—during link creation, a user does not have to specify the starting point of recording (it is implicitly given by the most recent shortcut-reachable view), and, during link invocation, replay happens in the background to give users a true click-and-go experience.

Another challenge uLink addresses is reducing developer effort. uLink is implemented as a library that developers include in their app. The library transparently interposes UI events and data dependency between views with minimal developer effort. For shortcut-only links, which are the only ones existing mobile deep links can support, uLink needs only one line of code *per page class* of the app (as opposed to tens of lines of code per deep link in existing mobile deep links). For shortcut-and-replay links, which existing mobile deep links do *not* support, developers need to write one line of code *per UI event handler*. Our evaluation with existing apps shows that the overall effort is minimal.

The final challenge uLink addresses is identifying links that may not open correctly at some later point in time. This is not surprising since even a full-featured record and replay tool may not guarantee reproducibility of the target view due to many nondeterministic factors. Broken links are common in the web as well. A link may not open correctly e.g., if the target view opens a file that is deleted after the link is saved, if a user is logged out from the app, or if some UI events cannot be captured or replayed (e.g., Android does not provide APIs for applying long taps on list items). However, it is important that the user gets a consistent experience—when she bookmarks a view, she should know whether she will be able to open the saved link in the future or not, and if not, why it might fail. A key contribution of this paper is to develop efficient techniques to provide such user feedback when a link is saved or opened.

We have implemented uLink as an Android library and evaluated it with 34 (of 1000 most downloaded) Android apps. While existing mobile deep links require in the order of 20–30 LoC *per deep link*, uLink can support shortcut-only links (a superset of deep links) to *all* views in an app with an average of 8 LoC *per app*. Overall, uLink achieved 70% links coverage and provided accurate user feedback (especially for links with file system dependencies). We also found that app links remain reasonably stable over time, with new app versions and updates in app contents.

In summary, we make the following contributions. (1) We develop uLink, a mobile app deep linking mechanism that requires much less developer effort, but provides significantly more coverage than existing mobile deep linking. (2) We develop techniques to predict if an app link may break in the future, and, if so, under what conditions. (3) We evaluate uLink with 34 real apps.

2. MOTIVATION AND GOALS

In this section, we motivate the need for uLink, set our goals, and review related work.

2.1 uLink in Action

We motivate uLink with a few concrete scenarios. Our goal here is to demonstrate some of what uLink can enable. We leave details of its design, challenges, solution, and implementation to next sections of the paper.

Developer experience. For ease of deployment, uLink is implemented as a user-level library, similar to existing analytics libraries, such as Localytics [30], Flurry [18], and Appsee [8]. The developer includes the uLink library in the app, and this alone readily makes the app uLink-enabled, with shortcut-only links (Figure 1(a) and 1(b)). To enable shortcut-and-replay links (Figure 1(c)), the developer adds one line of code in every UI event handler of the app.

uLink library and companion services. As the user uses a uLink-enabled app, the uLink library continuously tracks explicit data flow between app views (e.g., intents transferred between Android activities) and UI events. At any point in time, the user can request a link to the current app view, e.g., by shaking the device in our current implementation. An external companion service, typically a first-party service, can request to be notified each time a user requests a new link or it can request uLink to automatically create links to all views the user visits within specific types of apps (e.g., all third party apps). Two such companion services we implemented are: Bookmark, which stores all links the user wishes to save and invoke later; and Stuff-I've-Seen, which indexes contents and links to all app views the user visits and, like a web search engine, allows the user to search app content and directly navigate to content of interest using the associated link.

User experience. Here are few scenarios a user can experience with a uLink-enabled app and companion services.

(1) In a recipe app, the user can bookmark any page containing her favorite recipes. She can later invoke the saved links from the Bookmark service to directly open each recipe page.[1]

(2) She can create macro-like links for frequent tasks in an app. For example, in a library app, she can go to the "renew book" page,

[1]In theory, these bookmarks can also be shared with friends, in the same way we share web links.

System	Dynamic links	Convenience	Coverage	Dev effort	Ease of deployment
Web URLs	Yes	High	Good	None	Yes
Web macros	Yes	Low	High	None	Yes
Mobile deep links	No	-	Low	Some	Yes
App record&replay	Yes	Low	High	None	No
uLink	**Yes**	**High**	**Good**	**Little**	**Yes**

Table 1. uLink goals and comparison with the state-of-the-art.

select all books, extend the return date by one more month, and finally hit the "renew" button, and create a link to capture the whole sequence of actions. Later, she can invoke the saved link (from the Bookmark service) to renew all her books with one single click.

(3) The Stuff-I've-Seen service runs in the background to index contents of all app views the user visits (recipes, news articles, hotels, etc.) along with their links. She can use the service to search previously-seen content, such as a recipe she read in some app in the past, and click on the link of the result to directly open it in the app.

For implementations of these scenarios see §4.2.

2.2 uLink Goals

Our overall goal is to enable mobile deep links that are similar to web deep links in terms of functionality and convenience. To provide useful and user-friendly links, uLink should satisfy the following requirements: **(1)** A user should be able to create a link *dynamically*, like web links, by specifying only the target app view. **(2)** uLink should have *good coverage*—a user should be able to create links to most, if not all, views of an app. **(3)** Invoking a link should be *fast*, and it should produce a *consistent* app view, despite minor changes in the app or its contents.

To be practical and reach a large population of smartphone users, **(1)** uLink should require *minimal developer effort* to make an app uLink-enabled. **(2)** uLink should incur *minimal runtime overhead* and have no impact on the app experience. For *ease of deployment*, uLink should not require changing the mobile OS or rooting the device. **(3)** uLink should be *compatible with existing mobile deep links* so that the underlying OS can offer the same user experience. For example, if a user generates a link for an app and later uninstalls the app and invokes the link, the OS will notice that the target app is missing and will redirect the user to the app store to install the app (this procedure is currently in place for deep links).

Ideally, uLink must capture *correct* links. Our notion of correctness tries to be as close as possible to that of today's web links. To illustrate, we classify web links into three broad categories based on how they behave.

- Client-side **immutable** links navigate to contents that remain the same, across devices, users, time and location. For instance, http://www.dictionary.com/browse/uri?s=t *always* points to the definition of the word "uri".
- Client-side **mutable** links navigate to contents that may change depending on: (1) *user* (e.g., a link to amazon.com page showing personalized recommendation), (2) *time* (e.g., a link to a news web site showing current top stories), (3) *location* or other sensors (e.g., a link to a weather web site showing local weather), and (4) the *device* (e.g., a link to the html5 web site https://www.picozu.com/ editing an image from the local file system).
- **Broken** links fail to navigate to any useful web page. This can happen if the original web page has been removed or renamed. A mutable link can break too, e.g., due to unavailability of some resources it depends on. For instance, a Facebook URL, saved on a device where the user is logged in, may fail when opened on a device where the user is currently logged out.

In other words, web links are best effort—they may break or may lead to contents that are different than the original contents. uLink provides a similar best-effort experience for app links. Many app links generated by uLink will be immutable and will always navigate to the same content. At the same time, some app links will be mutable and may show different contents based on current location, time, user, and device. For instance, an app link to the "Daily Top Stories" view of a news app will show different news stories every day.

We also note that it may not always be obvious to a user whether a link is mutable or not, and whether she should expect the content to remain unchanged. Consider the link to the app view in Figure 1(b). Since the location information ("New York") used by the view is provided by the previous view and is captured by uLink, the link generated by uLink will always show forecast for New York, irrespective of the user's current location. In contrast, had the app been written such that the current location was acquired dynamically by view Figure 1(b), the content of the link would have been different at different locations. Such ambiguity, however, is also common for web links. For instance, both the links www.foreca.com/United_States/Washington/Seattle and https://www.wunderground.com/ show weather information; but the first link always shows the weather of Seattle, while the second one shows the weather of the current location. We expect users to adapt to mutable app links in the same way they adapt with their web counterparts. For more discussion, see §7.

Finally, as in the web, app links may also break. uLink promptly detects when a link cannot be safely saved or replayed, and provides detailed feedback to the user or to the application capturing such links on the user behalf. Fundamentally, uLink cannot guarantee 100% coverage of an app's views because of the deployment constraints (minimal development effort and ease of deployment) it must satisfy. In §3.3, we enumerate uLink's possible failure cases, and explain what the feedback contains.

2.3 Related Work

We compare uLink with other related approaches, and explain why they are not sufficient to achieve our goals. Table 1 summarizes the discussion below.

Mobile deep links. As mentioned in §1, existing mobile deep links require nontrivial developer effort, have poor coverage, and are statically defined; therefore they do not satisfy many of our goals. We now elaborate on these limitations.

Mobile deep links require *nontrivial developer effort*. As an example, the open source Wikipedia app for Android has one deep link, and it requires 23 LoC to handle the associated intent. As a consequence, a small number of mobile apps, even among the top ones, expose deep links. An estimate by URX from 2014 says that 19% of the top 100 Android apps expose deep links (and only 11% have deep links for Android, iOS and iPad [40]). To confirm, we analyzed 13,848 Android apps downloaded in the month of May 2015 and covering all app categories.[2] Figure 2 reports the CDF of the number of deep links across apps.[3] 75% of the tested apps

[2]We counted deep links by looking in app manifest files for declarations of intent filters complying with the Android specifications for deep links [3]. Our counts can be over-estimates, since we did not verify if the app actually supports the deep links.

[3]The CDF excludes 0.25% of the apps which have more than 20 deep links. The maximum was for a social forum app with 1503 deep links.

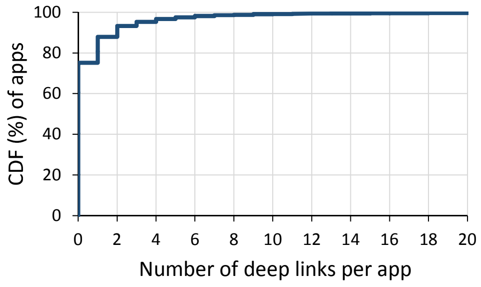
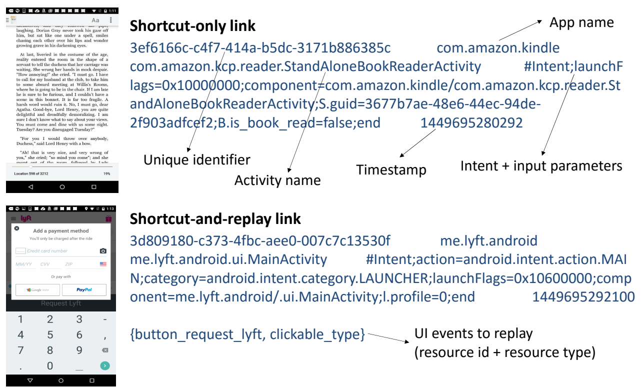


Figure 2. Number of deep links in top 14k Android apps.



```
3ef6166c-c4f7-414a-b5dc-3171b886385c          com.amazon.kindle
com.amazon.kcp.reader.StandAloneBookReaderActivity          #Intent;launchF
lags=0x10000000;component=com.amazon.kindle/com.amazon.kcp.reader.St
andAloneBookReaderActivity;S.guid=3677b7ae-48e6-44ec-94de-
2f903adfcef2;B.is_book_read=false;end          1449695280292
```

```
3d809180-c373-4fbc-aee0-007c7c13530f          me.lyft.android
me.lyft.android.ui.MainActivity          #Intent;action=android.intent.action.MAI
N;category=android.intent.category.LAUNCHER;launchFlags=0x10600000;comp
onent=me.lyft.android/.ui.MainActivity;l.profile=0;end          1449695292100

{button_request_lyft, clickable_type}
```

Figure 3. Examples of shortcut-only and shortcut-and-replay links.

do not expose any deep link; confirming the URX estimate of less than a quarter of apps having deep links.

Existing mobile deep links have *poor coverage*—a small number of locations, predefined by developers, within an app are directly accessible via deep links. As Figure 2 shows, 92% of the apps supporting deep links (23% of the total apps we analyzed) expose 5 or fewer deep links, indicating a small coverage. This is due to two key reasons. Developer's effort to support deep linking increases almost linearly with the number of unique deep links, and hence apps tend to expose very few deep links. It is unlikely that developers will open up all possible deep links in the app. Another more fundamental reason is that while deep links are stateless, mobile apps are stateful—an app's current view may depend on data from a previous view (e.g., location selected in a previous view) or as a result of specific user interactions (e.g., doing a search, selecting an item) on the current view. Thus, reaching the view may not be possible without transferring states from previous views or without applying the UI interactions. The stateful nature of apps also implies that even if deep links are free and a developer includes deep links to each and every page in an app and with a large number of supported input parameters, existing deep links, due to their statelessness, would not be able to capture and preserve user data generated in an app during interaction.

Finally, today's deep links are *defined statically* by developers, and must be known in advance to navigate to a target view; the dual process of dynamically determining the link for a given view is not possible even if a deep link to that view exists.

Record and replay. Record and replay techniques can record macros that can later be replayed to navigate to an arbitrary location of an application for desktop, server [14, 21, 31, 33, 36, 44], web [4, 24, 28, 29, 34], and mobile platforms [19, 23]. One might consider using such macros as links. A full blown record and replay mechanism can generate dynamic links and can have very good coverage, but it is not suitable to be augmented to support links for several reasons. First, these systems are too heavyweight to be used in the wild. Recording and replaying all sources of nondeterminism has prohibitive costs on mobile devices [17] (e.g., special hardware support or virtual machine instrumentation). There are tools that successfully record and replay smartphone apps by relying only on the sensor and UI event streams [19, 23], but they are still heavyweight and they require either a rooted phone or changes to the mobile OS, thus limiting their applicability to consumers' phones, at scale. Second, record and replay can be slow, especially when the user uses the app for a long period of time before arriving to the current view, and hence the replay phase needs to replay many user interactions. Finally, existing record and replay tools require the user to explicitly specify when a recording starts and ends—an unacceptable user experience when a user dynamically wants to generate a link to the current app view (e.g., for bookmarking it).

2.4 uLink Approach

We now briefly describe how uLink achieves the goals listed above; the details will come in the next section. uLink is implemented as a library that a developer includes in the app with *minimal effort*. The library continuously monitors various data dependencies and UI events of the current view so that, anytime, it can *dynamically* create a link with the dependencies encoded in it. Figure 3 shows two examples of links: the first link points to page 598 in a Kindle book, and the second encodes the sequence of actions for requesting a lift in the Lyft app (the result is the dialog for entering payment). After being saved, a link can later be invoked to *quickly* access the view, by taking shortcuts to views that depend only on data encoded in the link (e.g., book page), and/or by replaying, in the background, the UI events encoded in the link (a clickable button in the second link in Figure 3). This approach gives uLink *high coverage* of dependent views, which are not supported by existing mobile deep links. The overhead of shortcut-and-replay links is reduced by recording and replaying only UI events (button clicks, checkbox selections, etc.), which, as we will show later, is sufficient to recreate the target view with high fidelity, in many cases.

3. SYSTEM DESIGN

This section describes how uLink was designed to meet the aforementioned goals: high coverage and speed (§3.2), while ensuring minimal developer effort (§3.4).

3.1 Overview

Mobile apps consist of a set of *pages*, where each page typically contains a set of *UI elements* such as buttons, checkboxes, lists, or menus. Each UI element can have an associated *event handler*, which is invoked when the element is interacted with. The whole of a page may not be viewable to the user at once; we call a part of the page shown in the current screen a *view*. A page may contain many views: the default view is what is shown on the screen when the user navigates to the page (Figure 1(b)), and the user can navigate to a different view within the same page by UI interactions such as selecting a tab (Figure 1(c)), choosing a date from a date picker control, filling out a search box, or clicking on a search button. A UI interaction can also lead from a view to the default view of a separate page.

Mobile apps are stateful and pages/views can have state dependencies. A page can use some data generated in a previous page. For example, the page containing the view in Figure 1(b) needs the location selected by the user in the previous page (in Figure 1(a)).

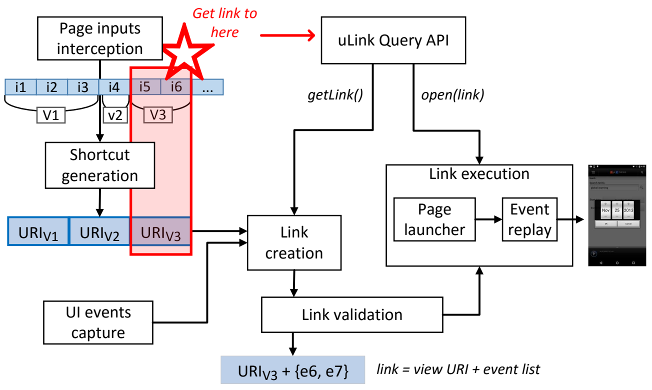


Figure 4. uLink system architecture.

A view can also depend on other views (e.g., one tab uses a flag set in another tab). Finally, a view can also depend on the sequence of UI actions starting from the default view. For example, the view in Figure 1(c) is obtained by tapping on the "POLLEN" tab in the default view.

Using this terminology, we identify three broad classes of links a user may capture in an app (listed in increasing order of complexity to support them). A user-defined link can link to the following states in an app:

1. *Stateless view*: A view whose state does *not* depend on states created in previous pages/views (e.g., a view showing weather, as in Figure 1(a)). It can be created without any input parameter, or with a set of statically-defined parameters that do not depend on previous pages/views.
2. *Stateful view*: A view whose state *depends* on app states created in previous pages (e.g., showing breathing forecast at a location *selected in the previous page*, as in Figure 1(b).
3. *UI-driven view*: A view created by UI events generated on the *same* page (e.g., Figure 1(c), created by tapping on the "POLLEN" tab in Figure 1(b)).

Existing mobile deep links support stateless views only. They can pass static parameters to target views, but cannot observe the internal state of the app (i.e., they live *outside* the app). This is precisely the reason why they cannot cover stateful or UI-driven views that depend on states (e.g., location selected by the user) and UI events (e.g., tapping on a particular tab) *inside* the app.

In contrast, uLink supports links to all the three types of views, and thus achieves its high coverage goal. Figure 4 shows the entire system architecture. In the following, we discuss the techniques we propose for creating, executing, and validating links.

3.2 Improving View Coverage

We first describe how uLink supports links for stateful and UI-driven views (not supported by deep links).

3.2.1 Links for stateful views

Stateful views depend on data from previous pages. By definition, a link to a stateful view points to the default view of its page.

uLink uses a novel technique called *shortcuts* to generate links to stateful views. We observe that a page in an app is usually instantiated through a launcher method responsible for rendering the page in the foreground (startActivity(intent,options) in Android and prepareForSegue:(uiStoryboardSegue) in iOS). This method usually expects as input a description of the page to render and possibly other parameters, which are not known to processes external to the app. This is equivalent to the query string in a web URL (e.g., in https://uLink.com/index.php?title=uLink_details&action=edit the query string is the part of the URI after '?').

Our key insight is that uLink can *program user-defined links by demonstration*: by observing how views are assembled during user interaction (V1, V2 and V3 in Figure 4), uLink can learn how to re-construct them. Specifically, uLink continuously intercepts all messages (i1, i2, etc.) sent to the page launcher method, so to infer message structures and input parameters necessary to render a view. uLink encodes the message structure and input parameters in a URI generated for the view (URI_{V3}). To open a saved link, the uLink library simply invokes the page launcher method with properly structured messages assembled using the parameters stored in the URI. In this way, uLink can *shortcut* to the default view of any page in the app. We call these shortcut-only links.

Assuming the link is opened under the *same conditions* (e.g., file system, sensors, and so forth) as when it was created, this approach guarantees accurate and safe, stateful links. They are accurate because this approach is goal-oriented. When a user requests capturing a link to the current view, the link is derived directly based on what app state was provided to *that* view. They are safe because this process does not risk breaking the program logic. We discuss later what happens if the link is opened under conditions different from those at creation time.

The above idea is simple and can be implemented by overloading the launcher method of the framework page classes (e.g., startActivity method of Android Activity classes).

3.2.2 Links for UI-driven views

The above technique of intercepting data passed between pages does not capture UI events within a page, and hence is not sufficient to recreate a UI-driven view. To support such views, uLink adopts a limited form of record and replay.

uLink continuously monitors UI events triggered during user interactions, and associated event handlers that are fired. To reduce overhead, uLink monitors UI events only in the current page; when the user moves to a different page, the UI events of the previous page are discarded. This approach does not compromise coverage since uLink can directly navigate to the default view of the current page by using a shortcut-only link to the (stateful) view. To create a link to a UI-driven view, uLink encodes two pieces of information in the link: (1) input parameters to launcher method of the current page (same as shortcut-only links), and (2) UI events that lead the user from the page's default view to the current view. When the link is invoked, uLink first launches the page's default view by using its input parameters, and then replays the UI events to navigate to the target view. The UI events are replayed in the background, and so the user sees the same click-and-go experience as shortcut-only links. We call such links shortcut-and-replay links.

Mobile app contents can be dynamic. For example, suppose a page's default view shows a list of restaurants. User clicks on the second item "Kabab Palace" to generate a view with the details and menu of the restaurant, and saves a link to the view. Now suppose, after a month, the app updates its contents, and the same restaurant "Kabab Palace" appears as the fifth item in the list. To deal with such changes, unlike traditional record and replay systems, uLink prioritizes content over UI structure during replay—it will search the list to find "Kabab Palace"; if it exists, uLink will click on it irrespective of its position in the list. Only if "Kabab Palace" does not appear in the list anymore, uLink will fall back to structural replay and click on the second item of the list. Such fallback is useful to deal with highly dynamic contents, such as a link to the top news story, which may change frequently.

We also observe that compared to record and replay tools, this approach does not require any recording start point, and it is much faster. Imagine a task where a user searches through old news by first entering some keywords and then specifying a date range, thus landing on the view with the news matching the specified criteria. The user wants to save a link to this view with the results. A standard record and replay tool (i) would require the user to specify the task's start point (i.e., the view where the keywords were entered), and (ii) would replay every single user action, i.e., entering the search string, clicking on the Start date button, pressing the search button, etc. Instead, uLink intercepts the inputs needed to generate the search result page and, through *a single* function call, loads the page. Then, only some, if any, UI events are replayed.

uLink does *not* need to record the time gaps between separate UI events. Instead uLink replays the sequence of recorded UI events such that each UI event is fired after the previous event has been dispatched, as notified by the device framework (e.g., on Android through the onUserInteraction callback). In this way, it is also possible to remove user-induced delays or idle periods.

On the other hand, uLink's record and replay is limited compared to existing record and replay systems: it captures only UI events (button clicks, checkbox selections, etc.), which, as we will show later, is sufficient in many cases. Capturing I/O and sensor access operations would bring us closer to the ideal world of deterministic replay, but monitoring these events would lead to unsupportable overheads in terms of annotations that developers would have to provide, in terms of OS modifications, or in terms of runtime overhead. By capturing only UI events, uLink hits a sweet spot between existing lightweight but low-coverage deep links and heavyweight but high coverage full-blown record and replay.

3.2.3 Limitations

Since uLink captures only data passed through page launcher methods and UI events within a page, there will be cases where uLink won't be able to correctly open a saved link at a later point in time. For example, consider an eReader app's link that opens a book (stored in the local device) at a specific page (saved in a configuration file). The link will fail if the book is removed from the device, or it may lead to a different page of the book if the page number in the configuration file is modified after the link is created. As mentioned before, taking a snapshot of all resources that a link might depend on can be prohibitively expensive.

To address this, uLink incorporates a feedback mechanism. Intuitively, uLink tracks all dependencies that it may fail to capture in the link. This gives users, or applications on their behalf, an idea about whether the link can be faithfully invoked in the future, and if not, why. Using the feedback, users can rely on their own judgment to decide how to use the link. Such feedback can also be useful to deal with link ambiguity (§2.2). For example, while bookmarking a link, uLink may notify the user that the current view depends on her location, which uLink is not capturing in the link. Knowing this, the user may or may not save the link in her favorites, depending on whether she expects to invoke the link from a different location.

3.3 Link Validation

uLink provides feedback to users (or applications on their behalf) at the time of link creation and of link execution. To be more concrete, suppose a user navigates from Page1 to Page2 of an app, and wishes to create a link to Page2. Can the link be correctly invoked later to see the same content? The answer depends on whether the pages access any external resource such as a file (outside the

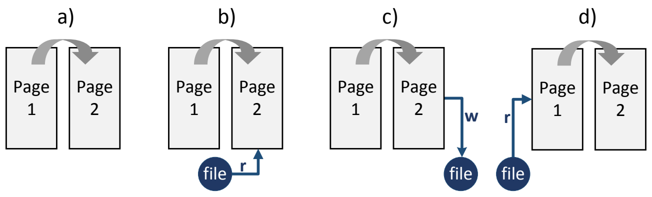


Figure 5. uLink can replay correctly a link to Page2 in case a), c) and d), and in case b) if the file doesn't change after link creation.

parameters explicitly transferred from Page1 to Page2's launcher method). Let us consider the four cases in Figure 5, with different dependencies from external resources.

Case (a) No external dependencies: uLink can correctly open Page2, by providing the parameters stored in the link to the launcher method of Page2.

Case (b) Page2 reads an external resource: (1) uLink may not be able to correctly open Page2, or (2) it may be able to open the page, but with potentially different content. This may happen if the content of the external resource is modified after the link is created. For example, if Page2 plays a specific music file from the local device, a link to the page will fail if the music file is deleted from the device. Similarly, if Page2 shows content of a local file, a link to the page will show different content if the file is modified after the link is created.

Case (c) Page2 writes an external resource: uLink can correctly open Page2, since the content of the external resource does not affect the content of Page2. So Page2 remains unaffected even if the external resource is modified after the link is created.

Case (d) Page1 reads an external resource: uLink can correctly open Page2. If the external resource somehow affects the content of Page2, its value must propagate through the data passed from Page1 to Page2, which uLink correctly captures. Note that, in order to capture possible changes to the external resource that Page 1 depends on (which may affect the value passed to Page2), the *reference* to the value (e.g., filename) and *not* the value itself should be passed from Page 1 to Page 2. Whether link arguments are passed by value or by reference is up to the application, and links behave accordingly.

Of all the cases above, Case (b) is the only case where uLink *may* not be able to correctly open a link to Page2. Ideally, preventing such behavior would require recording contents of all the external resources as well, which can be prohibitively expensive. uLink therefore does not try to prevent such behavior, and rather notifies users or the companion services at link creation or execution time that the link may not be replayed correctly, if a specific resource is modified. We call this process link validation.

Lightweight dependency tracking. The key challenge in supporting link validation is that it must happen during user interaction, at minimal overhead, and with minimal changes to the app. A possible approach, which requires no app changes, is to monitor I/O and sensors by instrumenting generic OS-provided APIs (e.g., system calls). However, this is not ideal because it needs instrumentation of the framework and incurs runtime processing and storage overhead. Another approach is to inject the monitoring logic in the app and track information flow in the app (similar to taint tracking [15]); however, this requires nontrivial development effort and incurs high runtime overhead. Our solution is to rely on an offline, automated analysis of the application to generate an app-specific *summary* of resource dependencies of relevant event handlers. Once the summary is built offline, it is installed on the device (by the uLink library downloading it from the cloud), and consulted each time a link is saved or opened. By design, this approach cannot be as accurate as heavyweight API instrumentation, taint tracking or other approaches requiring OS modifications, but it provides a first, practical approximation of the problem, while not compromising our goals of low overhead and minimal developer effort.

To frame the problem, we define the following terms. We collectively call *resources* entities external to the app that can change arbitrarily, after link creation. These include files, databases, and sensors. Our definition of "change" means any type of modification to the content or properties of the resource. For example, a file content can be "changed" by overwriting the file or by modifying one of the attributes such as read/write permissions. We call *source APIs* and *sink APIs*, the APIs the application framework provides to get or to put data, respectively, into such resources. In particular, we consider the following categories: file system read and write, database read and write, and sensor read operations. The list of such APIs can be obtained using tools such as SuSi [37]. Finally, we collectively call *callbacks* event handlers triggered, synchronously or asynchronously, by the app (e.g., in response to interacted UI elements) or by the framework (e.g., app lifecycle events).

Note that we do not include the network because, as with web URLs, we have no control on the app (or web site) backend, so we aim for link correctness under the assumption that the backend has not made the link's content unavailable (e.g., by removing the corresponding data objects nor by changing their identifiers, without providing a redirection).

Static analysis. Our first effort on generating offline app summaries is to rely on static analysis of the app. We generate the call graph of the app, and then recursively traverse it to find out a connected path from a callback to a source or a sink API. If we find any connecting path, we add the mapping <callback,source> or <callback,sink> to the summary. Online, each time a link is saved, the uLink library logs which callbacks have been invoked to generate the app state. In this way, uLink can use the summary to look up whether any of such link-required callbacks has dependencies on external resources, and report that in the feedback. Note that once the uLink library is added to the app, the developer effort required for the feedback generation is zero, because all link-required callbacks are logged directly by the library.

This approach proved relatively robust, meaning that we did not encounter any case in which a resource that was accessed by a link was not caught by the summary. However, we found it too conservative because of its coarse-granularity. For example, knowing that a specific callback reads *some* file is not sufficient to infer whether this operation may or may not compromise the link. If that file doesn't change after link creation, then the link will work correctly. If the file has changed, the link may or may not work. However, the problem is that online, with our restricted monitoring setup, we cannot capture the identifiers of the modified resources.

Static + dynamic analysis. To address this, we augment static analysis of the app with dynamic analysis. To exemplify, let us consider the file system resource. Our key insight is that while we do *not* know the file identifiers online, we can capture them offline and leverage them to establish *callback relationships*. We run each app using a Monkey (such as the Android [22] or Windows Phone [32] UI automation tools). We collect logs of all invoked callbacks as well as traces of file system accesses (using strace-like utilities). We then intersect read and write operations that share files with the same identifiers. The obtained file-relationships are added to the summary. For example, the dynamic analysis may produce a trace where callback c1 reads file f, and callback c4 writes to file f, so a c1 $\rightarrow$ c4 file-relationship is added to the summary.

Note that a perfect dynamic analysis would deem static analysis unnecessary. However, existing Monkeys have less than ideal coverage; we therefore primarily rely on static analysis to generate summaries for *all* callbacks, and refine the summaries with more fine grained information whenever they are available from dynamic analysis (i.e., whenever the Monkey exercises the event handler).

Once we have generated the summaries, we use them online in the following way. As the user interacts with her apps, the uLink library continuously monitors all callbacks invoked due to user interaction, and keeps a log *only* of the callbacks that led to a file system or database write. When a link previously saved is opened, the link-required callbacks (recorded when the link was saved) are processed using the summary. Any link-required callbacks associated with file system (or database) writes can be safely ignored (as in Case (c) in Figure 5). Instead, any link-required callbacks associated with file system (or database) reads can be ignored only if they have no relationships with other write callbacks that were logged after the link was saved. In fact, this means that re-creating the link state requires reading a file (or a database) that was modified in the past, after the link was saved (as in Case (b)). If such file (or database) dependency is found, uLink generates a feedback report including details about the identified root cause, i.e., the callback information and the source/sink API. Using the example above, suppose that the link-required callbacks of the saved link include callback c1, and that the callback log contains c4. Because c1 has a file dependency on c4, the link correctness may be compromised.

We cannot keep indefinite logs of all write callbacks, for all apps. We approximate by keeping a runtime log for a maximum period of time (currently an hour). Our intuition is that writes that are older than that period are likely to have been absorbed by the system (e.g., changes in preferences) so that they can be safely forgotten.

Our implementation currently provides fine-grained feedback only for the file system, while detects database changes only at the granularity of read/write (through static analysis). However, the same approach can be applied to databases by intercepting database APIs in the app framework (offline, during dynamic analysis). For sensors, fine-grained analysis of read/write operations is less critical because with the exception of a few sensors (e.g., microphone) sensor operations are always reads. This also means that links cannot break because of changes in sensor values. However, sensors can cause link ambiguity, which we discuss in §7. Finally, since it happens offline, the dynamic analysis could also be improved by adopting heavyweight tools such as taint tracking [15].

Other failure types. A captured link may fail also due to technical limitations of our system or of the app framework on which it runs. For instance, on Android, it is possible to track long tap UI events for items in a list, but the framework doesn't provide an API for replaying such events (while it does for single tap events). A link may also fail because of developer errors. Record and replay requires the developers to add one line of code per each UI event handler. If the developer forgets to instrument them all, a link may fail. Such kind of problems are captured by the uLink library when re-creating the link state, and feedback is provided.

3.4 Developer Effort

uLink is implemented as an application library. To make her app uLink-enabled, a developer includes the uLink library and extends the uLinkPage class provided by uLink, instead of the original Page class provided by the underlying framework (this is needed to overload the framework's page launcher method). Once the library is added, shortcut-only links are readily enabled.

To support shortcut-and-replay links, app developers must add one line of code in each UI event handler of the app. This effort

Change	Dev effort (LoC)	Enabled links
Add uLink library, extend from uLink-provided Activity	1 per main Activity class	shortcut-only links
Log UI event handlers	1 per UI event handler	shortcut-and-replay links

Table 2. Developer effort to add uLink in Android apps.

is larger, but still small (see §5.1). Moreover, since this process is rather mechanical, the logging statements could be injected automatically through a source-to-source transformation tool. Table 2 summarizes the developer effort for Android apps.

4. IMPLEMENTATION AND USE CASES

This section provides details on our Android implementation of uLink, and describes three companion services we have built.

4.1 uLink Library

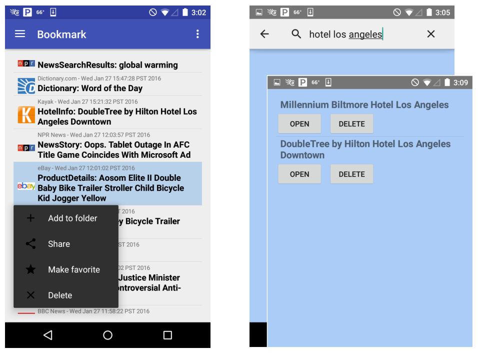
We have implemented uLink on Android 5.1.1. On Android, pages, called *activities*, are started by taking a direct Android *intent*. Activity classes are created by extending one of 13 Activity classes provided by Android. An activity can be launched by supplying an Android intent, which is essentially a passive data structure containing an abstract description of an action to be performed. To transparently capture the intents, so that a target activity can be directly launched by supplying the necessary intent, uLink uses the following tricks: (1) It provides one *uLinkActivity* class for each of the framework-provided Activity classes, with the same external interfaces, so that developers can extend the uLinkActivity class, instead of the framework-provided Activity classes, to create a new Activity class. (2) The launcher methods of the uLinkActivity classes are instrumented to dynamically capture the intents provided to them, and to encode them in URIs to the app views.

The above tricks could be used also in other platforms. In iOS, Scene extends NsObject.UIResponder.UIViewController, and, in Windows, Page extends Systems.Windows.Controls.Page. So the uLink library for those platform could provide replacement classes to be extended by developers to create app pages.

Developer effort. To enable shortcut-only links, each *main* Activity must extend the corresponding uLink-provided Activity class. By main Activity, we mean any Activity that implements one of the 13 Android Activity classes. App developers often create one or a few customized main Activity classes that extend from the framework-provided classes, and use them to instantiate all other activities. Hence, the overhead is rather small (see §5.1).

To support shortcut-and-replay links, uLink requires app developers to invoke the trackEventHandler(view, view_type) method of the uLink library, each time a UI event handler is fired. Developers provide the corresponding View object (i.e., UI Element) that raised the event and its type, such as button, list item or textbox. Note that for some types of UI event handler, such as click event handlers, this information could be captured automatically through the framework by probing the View objects inside an Activity (effectively its UI tree), when it is first loaded and assigned trackers. Since click event handlers are the most common type of UI event handlers, this would be a significant reduction in the developer effort for shortcut-and-replay links. We are currently exploring this approach. Table 2 summaries the uLink developer effort.

To provide summaries for link validation, for the static and dynamic analysis of Android apps, we generated the call graphs using the Soot [35, 41] analysis framework. Since Android applications do not have a traditional Main entry point, we created a dummy


Bookmark

NewsSearchResults: global warming

Dictionary.com - Wed Jan 27 15:47:28 PST 2016
Dictionary: Word of the Day

Kayak - Wed Jan 27 15:21:32 PST 2016
HotelInfo: DoubleTree by Hilton Hotel Los Angeles Downtown

NPR News - Wed Jan 27 12:03:57 PST 2016
NewsStory: Oops. Tablet Outage In AFC Title Game Coincides With Microsoft Ad

eBay - Wed Jan 27 12:01:02 PST 2016
ProductDetails: Aosom Elite II Double Baby Bike Trailer Stroller Child Bicycle Kid Jogger Yellow

BBC News - Wed Jan 27 11:58:22 PST 2016

hotel los angeles

Millennium Biltmore Hotel Los Angeles

DoubleTree by Hilton Hotel Los Angeles Downtown

Figure 6. Bookmark (left) and Stuff-I've-Seen (right) services we have built using uLink.

entry point leveraging the approach of Flowdroid [9]. We used the Android source-sink APIs listed by SuSi [37]. This list includes 26,322 source and sink APIs available from Android 4.2. We selected 2280 sources and sinks, and grouped them into six classes: file system, database, resources, media, camera, and sensors.

Query API. uLink provides the following API that an application or companion service can use to programmatically generate or invoke links. (1) getLinkToCurrentView(): Returns a link to the current view. (2) getLinkOnCondition(condition, callback): Registers a callback, which will be invoked with a link and content of the current view, when the current view matches a condition. For example, for the Stuff-I've-Seen service described below, the condition can be "when the view belongs to a third party app and the user has spent more than 5 seconds on it". (3) openLink(link): Opens the specified link. If the app is not currently installed, it takes the user to the app store to install it. The object link (viewURI, events, callbacks) contains the view URI, the list of events for replay (possibly empty), and the list of link-required callbacks (i.e., callbacks that were invoked when the link was first saved).

4.2 Companion Services using uLink

The following three services demonstrate different usages of uLink.

Bookmark. Being able to bookmark links to an arbitrary state in an app is useful for various purposes. (i) When launched, mobile apps always start from the same entry page. Even if a user *always* only cares about the content appearing on a certain page, say the third, she must always go through the first and second page. A user that uses such app every day, perhaps multiple times, would benefit from being able to create a shortcut to the target page. (ii) When interacting with an app, a user may find some content she wants to save for later. (iii) Most users have tasks that repeat identical every day or every so often, such as monitoring the price of an item to purchase, ordering a pizza or checking whether new interesting houses are on the market. These tasks are likely to always take the same user inputs (e.g., number and type of pizzas). Users would benefit from being able to record such repetitive tasks and automate their execution. (iv) Filling forms in mobile apps is notoriously painful. Having the ability to record user inputs (e.g., login information, search parameters, etc.) for specific app pages can improve user productivity. We built Bookmark (left-hand side of Figure 6) that collects links to content, actions, tasks a user wishes to save. Each time a user shakes her phone, a link to the current view is saved into the Bookmark. Links are opened by clicking on them.

Stuff-I've-Seen. Users browse lots of content inside their apps (e.g., hotels to book, restaurants to visit, news article to read), and

App	**Category**	**Description**	**Downloads**	**Total**	**uLink LoC**	
				LoC	**shortcut-only**	**shortcut-and-replay**
NPR News	News & Magazines	News reader	1M-5M	13,114	2	18
AnkiDroid	Education	Flash card manager	1M-5M	45,959	7	66
Book Catalogue	Productivity	Book list manager	100K-500K	41,587	8	228
Vanilla Music	Music & Audio	Music player	500K-1M	15,518	4	79
K9-Mail	Communication	Email client	5M - 10M	67,721	4	75
eBay	Shopping	e-Commerce app	100M-500M	500,251	6	368
Lyft	Transportation	Taxi service	1M-5M	356,894	1	30
Spotify	Music & Audio	Streaming music service	100M-500M	523,999	12	430
Amazon	Shopping	e-Commerce app	10M-50M	418,503	10	235
Amazon Kindle	Books & Reference	e-Book reader	100M-500M	432,040	13	346
BBC News	News & Magazines	News reader	10M-50M	398,835	12	170
Watch ESPN	Sports	Live sports	10M-50M	452,437	5	125
AccuWeather	Weather	Weather update	50M-100M	392,707	20	132
Aldiko Book Reader	Books & Reference	e-Book reader	10M-50M	239,967	6	198
ASTRO File Manager	Productivity	File manager	50M-100M	393,712	10	231
Photo Editor by Aviary	Photography	Photo editor	50M-100M	340,653	3	181
Booking.com Hotel Reservations	Travel & Local	Hotel reservations	10M-50M	317,012	5	491
APUS Booster+	Productivity	System utility	10M-50M	29,923	7	75
Compass PRO	Tools	Utility	5M-10M	167,415	3	68
Dictionary.com	Books & Reference	Dictionary software	10M-50M	388,963	32	229
Duolingo	Education	Language tutorial	10M-50M	253,975	2	190
Hulu Plus	Entertainment	Live streaming	100K-500K	411,272	11	204
KAYAK Flights, Hotels & Cars	Travel & Local	Hotel, flight, & car manager	10M-50M	380,609	6	390
MakeMyTrip-Flights Hotel	Travel & Local	Hotel, & flight	5M-10M	421,598	15	345
Dictionary-Merriam-Webster	Books & Reference	Dictionary software	10M-50M	217,013	3	71
Music Player for Android	Music & Audio	Music player	10M-50M	89,411	5	79
Retrica	Photography	Camera & photoeditor	100M-500M	266,428	8	172
SPB TV	Media & Video	Streaming TV	10M-50M	318,170	5	163
the Weather	Weather	Weather update	10M-50M	273,724	7	167
TuneIn Radio	Music & Audio	Streaming radio	100M-500M	340,260	19	276
Advanced Task Killer	Productivity	System utility	50M-100M	12,843	6	50
WebMD for Android	Health & Fitness	Health app	5M-10M	319,537	7	170
Yahoo!	News & Magazines	News reader	10M-50M	395,316	10	260
Zillow	Lifestyle	Apartment finder	10M-50M	404,068	11	344
			Average	**283,572**	**8.4**	**195.8**

Table 3. The 34 Android apps to which we added the uLink library and the developer effort required. (First 5 apps are open source.)

sometimes would like to be able to search through "all the stuff they have seen", and *not* through all the content those apps (or the web) offer. We built Stuff-I've-Seen (right-hand side of Figure 6), reminiscent of similar work for the web [13], for desktop applications [27], and for entire computers [11]. This service transparently logs contents the user sees in her apps, it indexes them, and provides basic search capability. The app contents (i.e., texts appearing in the UI elements in the app page) are obtained by processing the app's UI tree. For indexing, the service uses the Apache Lucene library [5]. Semantics analysis (currently not implemented) could also be performed locally [16] or in the cloud (as in Google's Now on Tap [20] and Bing Snapp [43]). The app currently tracks eBay product details, Kayak searches, NPRNews news, Spotify's songs and artists, and Kindle's books.

360-IFTTT. IFTTT [25] is a popular app that allows users to "program" <if-do> recipes such as "If it rains, remind me to take an umbrella". Currently these recipes are built using open APIs and web sites. With uLink, recipes can tap into third party apps. We built 360-IFTTT. Users can specify simple "if" conditions based on location and time. "Do" actions are specified by executing the desired task with the app (once), saving the link by shaking the phone, and copying the link from Bookmark into 360-IFTTT.

5. EVALUATION

In this section, we evaluate uLink on five metrics: *developer effort*, *coverage*, *correctness of link validation*, *link consistency over time* (i.e., whether the links remain valid over time and across app versions), and *runtime overhead and performance*. We report the results for both shortcut-only and shortcut-and-replay links. Our evaluation is based on 34 Android apps, shown in Table 3. We first tested the library with three apps (NPRNews, Lyft, eBay) and later integrated it into 31 more apps, *without* any changes to the library, confirming the generality and applicability of our approach.

5.1 App Dataset and Developer Effort

The uLink library was integrated successfully in a total of 34 apps. Among the top 1000 Android apps, we selected apps based on popularity and compatibility with Android 5.0 from a variety of app categories, with the exclusion of games (they are not in scope of our uLink-based scenarios) and native code apps. Five of the selected apps are open source, and hence we could modify their source code to include uLink. For the other apps, we used Soot [35, 41] for Dalvik bytecode instrumentation. The limiting factor in integrating uLink into closed source apps was not the complexity of the logic for injecting our changes, but the recurrence of bytecode obfuscation in apps. Once instrumented, we verified that they worked with 5 random links to 5 different pages.

The fact that we were able to integrate uLink through an automated instrumentation process is a first proof of how easy and mechanical the required changes are. In addition, we counted the lines of code (LoC) that were changed or added to integrate the shortcut-only or shortcut-and-replay variants of uLink. Table 3 shows the results. For comparison the table also reports the size of each app's codebase. To obtain an estimate for closed source apps, we counted the LoC after decompiling the app to Java source code using the dex2jar [12] and jd-gui [26] tools.The numbers do not give an exact count for LOC, but they provide a good approximation.

On average, shortcut-only required to change only 8.4 LoC in the app code. The smallest effort was 1 LoC (Lyft), and the largest 32 LoC (Dictionary.com). Recall that today's deep links can support only stateless links. Shortcut-only provides a superset of deep

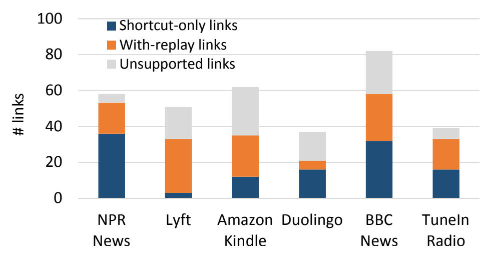


Figure 7. uLink link coverage in 6 apps (NPR News is open source, others are closed source).

links, with a much smaller developer effort. To give some comparison points, we inspected the code of two open source apps that support deep links. Ankidroid exposes one deep link which is handled in 35 LoC. Wikipedia also exposes one deep link coded in 23 LoC. With much less developer effort, uLink enables deep links to most app views (as we will show in the next experiment), and preserves application state.

The developer effort for shortcut-and-replay is higher (196 LoC on average) because it depends on the number of UI event handlers in the app, but the changes are still relatively few (on average 0.07% LoC of the entire codebase needed to be changed) and are rather mechanical. As discussed in §4, in the Android framework, at least the very popular click event handlers could be handled automatically.

Needless to say, if the uLink library could be added to the Android framework, it would require zero developer effort.

5.2 Coverage

We evaluate whether uLink can provide high coverage of an app views. We picked 6 apps, and manually enumerated all possible views in them. We were careful to report only unique views (e.g., if the same menu for adjusting screen zoom appeared in three different views, we counted it as one link rather than three). Then, we manually saved links to every such view, and opened them to verify whether the result was correct. In these tests we did not vary the operating conditions (e.g., same file system), so if links failed it was because of technical limitations of uLink or of the app framework.

Figure 7 reports the results. Across the 6 apps we found that on average there were 55 views one may save in a link. uLink provided coverage for 71% of them. Compared to the state-of-the-art, where if apps have deep links it is no more than a handful of links, this is a significant improvement. In particular, shortcut-only alone (which requires a tiny developer effort, see Table 3) provided an average of 19 links per app, and successfully enabled links to almost all pages' default views in the tested apps. Hence, with a much smaller developer effort, uLink provides much higher coverage.

The unsupported links were mainly due to failures in replaying UI events. Most failures were an artifact of binary instrumentation. In fact, for NPR News, the only open source app here, the coverage was 91%. In closed source apps, instrumentation failed to log custom UI event handlers (in the real world, the developer would provide the correct annotations), so some UI events (although captured) could not be replayed. Other reasons for link failures were the following: i) there were UI elements to which the developer did not assign a resource identifier so they could not be replayed (this was the reason for the 9% failed links in NPR News), ii) there were some special UI events (e.g., list long click) for which the Android framework does not provide a replay API, and iii) there were page views that were displayed in a browser (in Kindle) which could not be reached by uLink.

Overall, uLink provides good coverage. With the framework's support we are optimistic this coverage can be almost 100%.

5.3 Correctness of Link Validation

To evaluate whether uLink can discover external dependencies of a saved link and correctly report when a link may fail, we conducted a controlled experiment with 16 links in 11 apps, with dependencies on file system, sensors, and databases.[4] The links emulate what a real user would like to save in such apps, such as shortcuts to relevant pages (e.g., hourly news, product pages) or to recurrent actions with saved inputs (e.g., refill a prescription, locate nearest radio stations, etc.). To verify the dependencies reported by uLink, we changed the resources after the links were created, and examined whether each link could open the original app view. For instance, in the Vanilla Music app, we saved a link for playing a song stored in the SD card, deleted the song (from the app) and opened the link. Each link was opened twice: (1) in the same conditions: after creating the link, we interacted with the app for a while and then opened the saved link, and (ii) in altered conditions: after creating the link, we forced a change in the resource(s) that the link depended on and opened it.

Table 4 shows the results. In analyzing them, recall that uLink currently monitors file system dependencies at fine-granularity, but database dependencies at coarse granularity. The table reports the ground truth and the uLink's output for both conditions. "Yes" means that the link can be safely opened, "No" that the link won't work, and "Maybe" that the link may not work (if the resource it depends on is modified). For No and Maybe, uLink provided a feedback on the root cause. Overall, uLink was wrong in only 2 out of 26 cases (marked in bold in the table). In 75% of the cases it agreed with the ground truth, and in the remaining 19% of the cases it took a conservative decision, mainly due to the lack of details on database read/writes. For instance, the "open a Kindle book" link requires reading the database. As uLink cannot yet track whether those reads have been affected by previous writes, it fires a warning. For links requiring reading the file system, uLink was more accurate. For instance, the open photo link in Aviatar requires reading a file (the photo). In the same conditions, uLink correctly detected that the link would work. In the altered conditions, it captured that the photo had been deleted and that the link-required callbacks had a dependency on that operation, thus reaching the correct conclusion. The feedback provided in No/Maybe situations was generally accurate – it was complete in 88% of the tested cases.

Overall, uLink's feedback is relatively accurate. With the addition of fine-grained database analysis, we expect the accuracy to be close to 100%. The services consuming app links can further decide how to interpret this feedback, especially in Maybe situations. For instance, "Stuff-I've-Seen" may be less conservative, and treat Maybe verdicts as Yes. Another service for automating recurrent user tasks (e.g., refill prescriptions) may be more conservative, and treat them as No.

5.4 Consistency over Time

We now explore whether uLink generates links that are reliable over time, in the face of i) app updates and ii) app content changes. To evaluate consistency across different app versions, we downloaded several older versions of some of our apps and tested

[4]For clarity of analysis, we further distinguished between read and write operations on preferences and cache. These resources can be easily recognized as their names remain constant across apps.

App	Link description	Dependency	Same conditions			Altered conditions		
				Ground truth	uLink		Ground truth	uLink
Amazon Kindle	Open a book page	r db, r pref	same db & pref	Yes	Maybe (r db, r pref)	del db, w pref	**No** (r db, w pref)	**Maybe** (r db)
Amazon Kindle	Sync books with cloud	w db, w pref	same db & pref	Yes	Yes	del db, w pref	Yes	Yes
Photo Editor by Aviary	Open a photo	r file	same files	Yes	Yes	del files	No (r file)	No (r file)
Lyft	Share referral code	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Lyft	Request a lift	loc	same loc	Maybe (loc)	Maybe (loc)	different loc	Maybe (loc)	Maybe (loc)
NPR News	Open a story	r/w cache in db	same cache	Yes	Yes	del cache	Yes	Yes
NPR News	Add news to playlist	r/w db	same db	Yes	Maybe (r/w db)	del db	Yes	Maybe (r/w db)
NPR News	Locate nearest station	loc	same loc	Maybe (loc)	Maybe (loc)	different loc	Maybe (loc)	Maybe (loc)
eBay	View product	r db	same DB	Yes	Maybe (r db)	del db	Yes	Maybe (r db)
WebMD	Refill a prescription through Wallgreens	camera, r pref loc	same loc same pref	Maybe (camera, r pref, loc)	Maybe (camera, r pref)	different loc, del pref	Maybe (camera, r pref, loc)	Maybe (camera, r pref)
Vanilla Music	Play a song	r file, w pref	same files & pref	Yes	Yes (r file)	del files, w pref	No (r file)	No (r file)
AnkiDroid	View a card	r db, r pref	same db & pref	Yes	Maybe (r db, r pref)	del db, w pref	**No** (r db, r pref)	**Maybe** (r db, r pref)
Book Catalogue	Reset hint	w db, w pref	same db & pref	Yes	Yes	del db, w pref	Yes	Yes
Book Catalogue	Manually add a book	w db	same db	Yes	Yes	del db	Yes	Yes
Dict. Merriam-Webster	View word of the day	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Dictionary.com	Search a word	w pref	same pref	Yes	Yes	del pref	Yes	Yes
Summary	**Decision (Yes, No, Maybe):**		**Wrong: 2 (6%)**	**Same as ground truth: 24 (75%)**		**Conservative: 6 (19%)**		
	No/Maybe feedback accuracy:		**Complete: 23 (88%)**	**Incomplete: 3 (12%)**				

Table 4. uLink feedback for links with dependencies on sensors, file system, and database (r=read, w=write, db=database, pref=preferences).

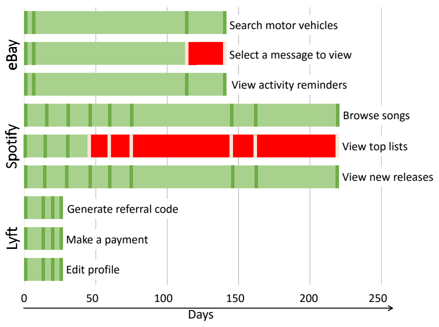


Figure 8. uLink across different app versions over time (green = link opened correctly, red = link failed).

whether links saved in the oldest version could be opened correctly in the newer versions. We tested 3 different links in Spotify (9 versions covering 7 months), in eBay (4 versions covering 4.5 months), and in Lyft (4 versions covering 1 month). Figure 8 reports our findings. For Spotify, 2 out of 3 links worked for the entire period; one link broke after the 3rd version update. Upon investigating further, we found that the cause was the removal of a UI element in the page layout. The second failure we noticed was for an eBay link which did not work for an intermediate version. In this case, uLink could not load the UI element and the app showed a dialog instead, but there was no crash. This study shows that links are relatively stable over a short period of time—there was no single failure for 50 days after link creation.

Then we investigated how uLink copes with updates to app contents. We created 10 links from 5 different apps (NPR News, Vanilla Music, Spotify, Book Catalogue and eBay). We selected common links such as viewing an eBay item, viewing the top NPR news, playing a song in Vanilla Music, or viewing the top releases in Spotify. Then for 4 weeks (8 weeks for NPR News), at least once a day, we ran a script that would open each of the links, and save a screenshot of the resulting view and debug logs. By inspecting the saved screenshots and logs, we observed that all 10 links, except 1, provided correct results for the entire duration of the experiment. Note that *some of the content were no longer available in the app itself; but the app backend still maintained the content and hence uLink could retrieve it*, highlighting the benefit of uLink. The failed link was for viewing a specific top news in NPR News. After 4 weeks, the link returned an empty page: the news item was not available anymore in the app backend with the same identifier.

5.5 System Overhead and Performance

The uLink library is 245 kB bytes big, so it adds a small storage overhead to existing apps. The computation overhead is also minimal, and processing occurs asynchronously to the app execution with no app slowdown. Generated app links are typically 100–150 bytes (e.g., the average size of the links used in Table 4 is 136 bytes).[5] For shortcut-and-replay links, uLink needs to log UI events. We executed AnkiDroid, a relatively I/O intensive app, for 10 minutes, and measured a callback log size of only 432 kB. With an hour time windows the cost is a few MBytes. We conclude uLink is a lightweight system.

uLink is also fast in opening saved links. When opening a link, if the app targeted by the link is already running (in foreground or background), the delay is the page rendering time. If the app is not running, uLink needs to first start the app and then render the page, so the total delay consists of the app loading time (typically 1–2 seconds depending on the app) and the page rendering time. We executed a simple experiment to verify that uLink's overhead on an app page's rendering time is negligible. We took 5 apps and created 3 links for each. We measured the uLink page rendering overhead as the difference between the 1) the average time necessary to open each link and achieve a stable page in the uLink-enabled app, and 2) the time the unmodified app takes to load the same page. As we were interested in comparing the page rendering times, we kept the app running in both conditions, but ensured the cache was cleared after each test. Moreover, when measuring the uLink rendering delay, we ensured the Activity targeted by the link was cleared on the stack (i.e., when the link is opened, we force a call to OnCreate). To measure the rendering time, we measured the time between the first call of OnCreate and the time when the root view of the Activity was inflated (i.e., all UI elements in the page are rendered). Across the 15 tests, the mean delay was 28 ms, so negligible.

In the previous test, we considered shortcut-only links. To evaluate the delay of shortcut-and-replay links we used RERAN [19], a record and replay tool for Android, as a baseline. Table 5 shows three example tasks with three different apps. We recorded the tasks (saved links) with both systems, stored the necessary logs, and then replayed each task (opened the link). We report execution time, number of replayed UI events, and logs size. Tasks are ordered by increasing number of clicks. uLink is from 5 up to 13

[5]The size of a link mainly depends on the size of the intent object captured by input interception. According to Android guidelines, this is maximum 1 MB. However, to keep links small, rather than saving large objects in the URI, the objects can be saved in storage and their reference be included.

App and link description	Replay time *(ms)* (& UI events)		Replay log size (*bytes*)	
	RERAN	uLink	RERAN	uLink
TuneInRadio: Stream a radio station	5,351 (4)	1,105 (1)	7,480	651
VanillaMusic: Go to a song and play it	7,762 (4)	982 (1)	6,880	693
NPRNews: Search news for some dates	17,345 (5)	1,392 (0)	11,296	984

Table 5. Comparison of uLink and the record and replay RERAN tool in terms of replay time, number of replayed UI events and log size.

times faster than RERAN. More importantly, while in RERAN the replay time increases as the task's complexity increases (number of visited pages and clicks), with uLink it remains fairly constant. The RERAN log sizes are also at least 10 times larger—for uLink, the replay log is essentially the link itself, which is quite small (last column). Finally, with uLink the average replay time is 1.2 seconds, which is close to what users expect for page loading times.

Overall, these experiments show that uLink is a lightweight system and provides an acceptable user experience.

6. OTHER RELATED WORK

We have discussed related work in §2.3. In addition to that, there are various companies offering mobile deep linking solutions with different corollary services (for a summary see [6]). AppLinks [7] comes with the Facebook Index API to check whether links are able to be deep-linked, and the possibility to code custom API. Deeplink.me [10] provides the AppWords Network for discovery and monetization of deep links across apps. URX [39] crawls web pages and constructs deep links metadata to be leveraged by app developers. Moreover, it dynamically extracts information from app pages to understand user intent, and provide deep links to content of interest. All these approaches contribute to building effective deep linking to app content, but all of them require significant developer effort and provide statically-defined deep links. Hence, compared to uLink, they suffer from the limitations we discussed earlier: low coverage and no support for stateful and UI-driven views.

7. LIMITATIONS

We discuss some key limitations of our work.

Technical limitations. uLink can currently capture only UI elements that have a unique resource identifier associated. Moreover, we rely on the Android framework APIs to programmatically replay UI events. For UI events for which Android provides only the screen coordinates (e.g., random screen touches), uLink can only mimic that action across devices with similar resolutions. uLink cannot handle certain gestures such as swipe, pinch or zoom, because they do not have replay APIs. These limitations have little impact on most apps and on our use cases. However, they can be a problem for use cases involving maps, games, etc.

uLink relies on the intents transferred between app pages. If the syntactic structure of the program changes, mostly due to a new app version (or to different versions across devices), uLink will be unable to reflect those changes. This means that old links may stop working. Changes in app versions could be tracked with cloud support, so to automatically detect/update broken links.

Fine-grained app summaries. uLink currently provides fine-grained summaries only for file system operations. For the file system we were able to leverage OS utilities (strace). For databases it is necessary to instrument the Android framework and app APIs to track the information flow. Taint-tracking [15] can also improve the precision of our approach.

Link ambiguity. It is not always possible to correctly understand a user's expectations when saving a link. For example, after saving a link to the "Nearby restaurants" view, when the user invokes this link later on, does she expect to see the restaurants near the current location or near the location the link was recorded at? In the NPR News app, when clicking on the *first* news item in the list of "Hot Daily News", a view showing the selected news story is displayed. If a user saves a link to this view, does she want to save a link to that specific news article or to the top daily news (i.e., the first item in the list)? As discussed in §2.2, link ambiguity is not unique to uLink, and, as for web links, practice will help users understand what app links can or cannot capture, on a per app basis. However, the system can help identify ambiguous situations, and prompt the user for clarification. Our system-generated feedback is a first step towards this goal. In addition, as in the web, link titles can also give hints on what the link is storing (e.g., uLink/NearbyRestaurants vs. uLink/NearbyRestaurants/NE+5th+St+Redmond+WA).

Unwanted replay. How can we prevent users from buying the same Amazon item twice by mistakenly clicking twice the same app link? This kind of problems are unlikely to be fully resolved without help from the app developer (e.g., in the form of annotations for pages or UI elements which should be excluded from link capture or that should request user confirmation) or without extracting app contents and action semantics so the system can reason about link purposes.

Link sharing: security and privacy. Users can share web links. From a technical point of view, users can share also app links. However, to make this viable, uLink needs to address at least two problems. First, it needs to handle cases in which a user may receive a link for an app that is not installed or that was saved in an app with a different version than the installed one. Second, security and privacy issues must be addressed: What if the shared link is malicious? What if the shared link makes changes to the app's preferences? What if the shared link contains personal information, such as login information or home address? Link encryption, cloud-aided link security analysis, and user/developer opt-out policies can help alleviate these issues.

App content revisitation. By default, uLink always provides the most updated content that a link references (which is also the case for web links). Revisiting app content is currently not supported, but it could be enabled by relying on a caching infrastructure and diffing tools similar to what proposed for the web [1, 2, 38].

8. CONCLUSIONS

uLink is a novel approach to enable deep links in mobile apps. uLink is distributed as a small library that developers include in their apps with tiny changes. Compared to mobile deep links, uLink provides higher coverage of an app views with less developer effort. uLink goes beyond the state-of-the-art: it provides links that are stateful and that can be specified by a user on demand, and it achieves these benefits without incurring large resource overheads nor modifying the OS. Although usability is not a goal of our system, uLink provides the first elements towards that goal: fast experience, no specification of a session start point, and feedback for links that may not work properly. We implemented uLink on Android and used it with 30+ apps with promising results.

9. ACKNOWLEDGEMENTS

We thank our shepherd, Landon Cox, and the anonymous reviewers for their thoughtful feedback. We also thank Earlence Fernandes for early discussion on the uLink design.

References

[1] E. Adar, J. Teevan, and S. T. Dumais. Large scale analysis of web revisitation patterns. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1197–1206. ACM, 2008.

[2] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas. The web changes everything: Understanding the dynamics of web content. In *Proc. of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 282–291. ACM, 2009.

[3] Android Developers. Enabling Deep Links for App Content. http://developer.android.com/training/app-indexing/deep-linking.html.

[4] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web Navigation with the WebVCR. In *Proc. of the 9th International World Wide Web Conference on Computer Networks*, pages 503–517, 2000.

[5] Apache Lucene. http://lucene.apache.org/.

[6] AppIndex. App Deep Linking Guide. http://appindex.com/blog/app-deep-linking-guide/.

[7] Applinks. http://www.applinks.org.

[8] Appsee. https://www.appsee.com/.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[10] Deeplink. https://www.deeplink.me.

[11] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic systems. In *Proc of. the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[12] dex2jar. https://github.com/pxb1988/dex2jar.

[13] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff I'Ve Seen: A System for Personal Information Retrieval and Re-use. In *Proc. of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 72–79. ACM, 2003.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.

[15] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–407. USENIX Association, 2010.

[16] E. Fernandes, O. Riva, and S. Nath. My OS ought to know me better: In-app behavioural analytics as an OS service. In *Proc. of HotOS XV*, 2015.

[17] J. Flinn and Z. M. Mao. Can deterministic replay be an enabling tool for mobile computing? In *Proc. of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 84–89. ACM, 2011.

[18] Flurry. http://www.flurry.com.

[19] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: Timing- and Touch-sensitive Record and Replay for Android. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81. IEEE Press, 2013.

[20] Google. Now on Tap. https://support.google.com/websearch/answer/6304517?hl=en.

[21] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-level Kernel for Record and Replay. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208. USENIX Association, 2008.

[22] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps. In *Proc. of MobiSys*, pages 204–217. ACM, June 2014.

[23] Y. Hu, T. Azim, and I. Neamtiu. Versatile yet lightweight record-and-replay for android. In *Proc. of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 349–366. ACM, 2015.

[24] D. Hupp and R. C. Miller. Smart Bookmarks: Automatic Retroactive Macro Recording on the Web. In *Proc. of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 81–90. ACM, 2007.

[25] IFTTT. https://ifttt.com/recipes.

[26] JD-GUI. http://jd.benow.ca/.

[27] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. DejaView: A Personal Virtual Computer Recorder. In *Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 279–292, 2007.

[28] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[29] I. Li, J. Nichols, T. Lau, C. Drews, and A. Cypher. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 723–732, 2010.

[30] Localytics. http://www.localytics.com/.

[31] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, May 2005.

[32] S. Nath, F. X. Lin, L. Ravindranath, and J. Padhye. SmartAds: bringing contextual ads to mobile apps. In *Proc. of MobiSys*, pages 111–124, 2013.

[33] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. *SIGPLAN Not.*, 43(3):308–318, Mar. 2008.

[34] A. Safonov, J. A. Konstan, and J. V. Carlis. End-user web automation: Challenges, experiences, recommendations. In *Proc. of WebNet 2001*, pages 1077–1085, 2001.

[35] Soot. Soot: A Framework for Analyzing and transforming Java and Android applications. http://sable.github.io/soot/.

[36] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of the Annual Conference on USENIX Annual Technical Conference*, pages 3–3. USENIX Association, 2004.

[37] SuSi - Sources and Sinks. http://sseblog.ec-spride.de/tools/susi/.

[38] J. Teevan, E. Cutrell, D. Fisher, S. M. Drucker, G. Ramos, P. André, and C. Hu. Visual snippets: Summarizing web pages for search and revisitation. In *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 2023–2032. ACM, 2009.

[39] URX. http://www.urx.com.

[40] URX Blog. How Many of the Top 200 Mobile Apps Use Deeplinks? http://blog.urx.com/urx-blog/how-many-of-the-top-200-mobile-apps-use-deeplinks.

[41] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[42] VentureBeat. Imagine a web without URLs. That's what the mobile app world looks like now. http://venturebeat.com/2014/07/08/imagine-a-web-without-urls-thats-what-the-mobile-app-world-looks-like-now/.

[43] VentureBeat. Microsoft beats Google to the punch: Bing for Android update does what Now on Tap will do. http://venturebeat.com/2015/08/20/microsoft-beats-google-to-the-punch-bing-for-android-update-does-what-now-on-tap-will-do/.

[44] M. Xu, R. Bodik, and M. D. Hill. A Flight Data Recorder for enabling full-system multiprocessor deterministic replay. *SIGARCH Comput. Archit. News*, 31(2):122–135, May 2003.