All Your App Links are Belong to Us: Understanding the Threats of Instant Apps based Attacks

Yutian Tang
ShanghaiTech University, China

Yulei Sui
University of Technology Sydney, Australia

Haoyu Wang
Beijing University of Posts and Telecommunications, China

Xiapu Luo*
The Hong Kong Polytechnic University, Hong Kong SAR, China

Hao Zhou
The Hong Kong Polytechnic University, Hong Kong SAR, China

Zhou Xu*
Chongqing University, China

ABSTRACT

Android deep link is a URL that takes users to a specific page of a mobile app, enabling seamless user experience from a webpage to an app. Android app link, a new type of deep link introduced in Android 6.0, is claimed to offer more benefits, such as supporting instant apps and providing more secure verification to protect against hijacking attacks that previous deep links can not. However, we find that the app link is not as secure as claimed, because the verification process can be bypassed by exploiting instant apps.

In this paper, we explore the weakness of the existing app link mechanism and propose three feasible hijacking attacks. Our findings show that even popular apps are subject to these attacks, such as Twitter, Whatsapp, Facebook Message. Our observation is confirmed by Google. To measure the severity of these vulnerabilities, we develop an automatic tool to detect vulnerable apps, and perform a large-scale empirical study on 400,000 Android apps.

Experiment results suggest that app link hijacking vulnerabilities are prevalent in the ecosystem. Specifically, 27.1% apps are vulnerable to *link hijacking with smart text selection (STS)*; 30.0% apps are vulnerable to *link hijacking without STS*, and all instant apps are vulnerable to *instant app attack*. We provide an in-depth understanding of the mechanisms behind these three types of attacks. Furthermore, we propose the corresponding detection and defense methods that can successfully prevent the proposed hijackings for all the evaluated apps, thus raising the bar against the attacks on Android app links. Our insights and findings demonstrate the urgency to identify and prevent app link hijacking attacks.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**.

KEYWORDS

Instant app, deep links, Android

ACM Reference Format:
Yutian Tang, Yulei Sui, Haoyu Wang, Xiapu Luo, Hao Zhou, and Zhou Xu. 2020. All Your App Links are Belong to Us: Understanding the Threats of Instant Apps based Attacks. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, USA, 12 pages. https://doi.org/10.1145/3368089.3409702

*The corresponding authors: X. Luo (csxluo@comp.polyu.edu.hk) and Z. Xu (zhouxullx@cqu.edu.cn)

1 INTRODUCTION

Mobile apps are pervasive. Android, as the dominant mobile operating system, supports a wide variety and a large number of mobile apps. To provide integrated service to users, Android integrates various functionalities from different apps. The *deep link* is a mechanism in Android to allow such seamless web-to-app communications [4, 16]. A deep link is a universal resource identifier (URI) for app content, such as a specific `Activity`. For example, clicking a deep link (e.g., `yelp:///career/home`) on a webpage in the mobile Chrome, a user is automatically redirected to the Yelp app by Android. Here, Chrome hands over the control to the Yelp as the latter is more suitable for the task.

App Link. Despite the convenience, researchers also identified serious security issues in deep links [8, 9, 39]. A most significant hijacking example is an app can register another app's scheme with a deep link and deceive users into opening the malicious app. To prevent this, Android promotes a new type of deep link called *app link* since Android 6.0.

The target of the app link is taking users directly to a link's specific content in an app. App links [16] are different from deep links in two aspects: first, app links only support links with HTTP(S) (e.g., `http(s)://...`). However, deep links support customized schemes (e.g., `yelp://...`); second, the app link verification is enforced for enabling app links. Whereas, such verification is not enforced for deep links. To pass the app link verification, a developer must do the followings: first, a user has to claim the app link in the app's manifest file (`AndroidManifest.xml`); second, the user has to publish a *digital asset link (DAL)* on their domain. The rules and syntax for constructing DAL are presented in the official tutorial [17].

Instant App. Despite the efforts made by the Android security team and researchers [22, 28, 31], app links are still not as secure as expected, especially when it is under the context of *instant apps* [2]. Instant apps are recently promoted by Android to enable on-demand use of modules in an app without the installation of the entire app. To build an instant app, developers must split an app into modules and associate URLs with these modules. These modules are named as *features* in an instant app.

Each instant app consists of one *base feature module* and zero or more *feature modules*. The *feature modules* can access all public functions in the *base feature module*. Inside each module, there is at least one `Activity` that serves as an entrance to the module. This `Activity` is always associated with a URI, through which users can access the `Activity` and other services in the module. When a

user clicks the URI, the module is downloaded and its `Activity` is launched to the user.

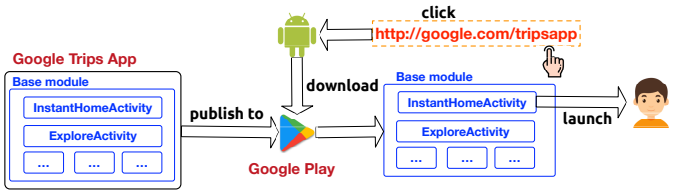

Figure 1: The Workflow of Android Instant App.

If a user accesses a *feature module*, an Android Package (a.k.a. APK, the binary format of an app) that contains the *base feature module* and the *feature module* is downloaded to the device. Otherwise, an APK that only contains the *base feature module* is downloaded [2] to the device. Fig. 1 shows an example of Google Trips instant app (`com.google.android.apps.travel.onthego`), which only contains the base module.

In Google Trips, there are two `Activities` that are associated with URLs. To be exact, `InstantHomeActivity` and `ExploreActivity` are bound to links `google.com/tripsapp` and `google.com/tripsapp/trip/em`, respectively. Once the link (i.e., `google.com/tripsapp`) is clicked, the module (base module in this case) is downloaded to the device. Then `InstantHomeActivity` is shown to users.

Compared with normal apps, instant apps have the following unique features: first, an instant app does not require any installation; and second, an instant app can provide on-demand modules for app users. A module is downloaded to the target device if and only if a user attempts to access certain functions inside that module.

Attacks. However, we find that app link verification can be easily bypassed by exploiting instant apps (detailed in Section 4). Attackers can manipulate a malicious instant app (MIA) to launch three types of attacks, including link hijacking with STS (§4.2), link hijacking without *smart text selection* (STS) (§4.3), and instant app hijacking (§4.4). Smart text selection (STS) is a novel feature introduced in Android 8.0 [36].

With STS, Android can recognize the text selected or tapped and recommend the next logical step. For example, if a user selects a sequence of numbers, STS recognizes them as a possible phone number and recommends the user to make a phone call. In this attack,the STS is spoofed to recommend our MIA for users. If a user select our MIA, they can be hijacked.

We validate these attacks on the latest Android versions (both Android 9 and 10). The attacks can be launched successfully on a Pixel device. We already reported the weakness of the app link mechanism to Google through its Vulnerability Reward Program (VRP). Google confirmed the vulnerability we reported. Our Android Id is 128919672.

To further measure the severity of these attacks, we develop a tool called *MIAFinder* to detect apps that are vulnerable to the aforementioned three types of attacks (detailed in Section 4). We then apply *MIAFinder* to 400,000 Android apps in the wild. Experiment results suggest that app link vulnerabilities are prevalent in Android's ecosystem, with over 30% of apps are fragile to these attacks (detailed in Section 6). To defense the app link attacks, we propose a novel API named *verifyDomainPackage* and generate a patch that integrates the *verifyDomainPackage* API for the latest Android (10.0). The experiments show that *verifyDomainPackage* can successfully prevent the all the three types of attacks (detailed in Section 7).

Contributions. The major contributions as follow:

- *New Vulnerabilities and Attacks.* We present the weakness of the app link mechanism in Android and present three kinds of attacks accordingly. These attacks are demonstrated to be practical and reproducible. For example, we demonstrate that our attacks can even hijack some popular apps, such as Gmail, Facebook Message, system SMS, and Whatsapp. To the best of our knowledge, this is the first end-to-end study of the security issues in the app link mechanism. Our observation is acknowledged by Google.
- *Detection and Defense Techniques.* We develop a static analysis tool named *MIAFinder* to automatically detect apps that are vulnerable to the three types of attacks. To defence these attacks, We further propose a defense mechanism and generate a patch for the latest Android 10. Experiment results show that our defense mechanism can successfully protect against these attacks.
- *Large-Scale Study.* We have conducted a large-scale empirical study on 200,000 apps from Google Play and 200,000 apps from Tencent Myapp, the largest third-party app market in China. Our results demonstrate that 53,619 Google Play apps and 54,650 Tencent-Myapp apps are vulnerable to *link hijacking with smart text selection (STS) attack*; 57,442 Google Play apps and 62,496 Tencent-Myapp apps are vulnerable to *link hijacking without smart text selection attack*; and all instant apps are vulnerable to *instant app hijacking attack*.

We hope that our efforts can raise awareness among relevant stakeholders (including smartphone vendors, app markets, app developers and mobile users). Hence, we have made our *MIAFinder* and all the experiments publicly available at: https://sites.google.com/view/instant-app-attacks.

2 MOTIVATING EXAMPLES

Architecture of MIA. We illustrate a malicious instant app (MIA) as shown in Fig. 3, which contains two parts, a phishing module and a benign module. When users use the MIA, only the functions in the benign module can be accessed. The benign module does not contain any harmful content, which makes users hard to recognize that the instant app is malicious. Thus the MIA hides itself as a non-MIA. However, once users access the victim URL, the phishing module becomes active and hijacks the URL. As Android hands over the control to the phishing module without notifying users, the users are not aware that they are interacting with a malicious app. As we successfully upload our MIA to Google Play (see demo videos on the project page), our MIA can bypass the security checking from Google Play and launch attacks without users' consent.

Link hijacking with STS. As shown in Table 1, the attack relies on a novel feature, *smart text selection (STS)*, which is introduced in Android 8.0. Once a user selects a piece of text, Android can recognize the text selected and then recommend the next logical step for the user. For example, when a user selects a sequence of numbers, STS may suggest the user make a phone call with the numbers

		Link Hijacking with STS	Link Hijacking without STS	Instant App Hijacking
Attack Vector		Instant app	Instant app	Instant app
App Link Verification?		Pass/Fail[1]	Pass/Fail	Pass
Victims	**in-app browsers/ browsers**	✓	✗	Nil[2]
	none in-app browsers	✓	✓	Nil
	instant app	Nil	Nil	✓
Assumption & Setup		We assume that a user installs a MIA from Google Play. For three different attacks, the settings for the MIAs are shown in Sec.4.3, Sec.4.2, and Sec.4.4 respectively.		
Attack Overview		When a user selects a text whose content is a URL, Android suggests the MIA to respond to it. For example, if a user select `www.yelp.com/biz/...` in an app (e.g, SMS app), The STS suggests our MIA to the user.	When a user clicks a link (e.g. www.yelp.com/biz/...), (s)he is redirected to the MIA. For example, if a user click the link in an app (e.g, SMS app), the STS suggests our MIA to the user.	When a user selects a link to launch an instant app, Android launches the MIA. For example, if a user clicks `google.com/tripsapp` to launch Google Trip instant app, (s)he is redirect to our MIA.
Countermeasures		To prevent this attack, developers can prevent users from selecting a text or stopping the STS. we propose 4 solutions for developers to prevent this attack (§5).	This attack can be prevented by implementing an in-app browser. Once a user clicks a link, the in-app browser can respond to the link.	The protection must be taken by Android. Android can leverage `verifyDomainPackage` (§7) to detect MIA.

[1]Pass: app link verification passed; Fail: app link verification failed; [2]Nil: this field is unavailable; [3]MIA: malicious instant app;

Table 1: Overview of Attack Models

selected. In STS, there are five functional texts for recommendations, including email address, phone number, physical address (in a longitude and latitude format), URL and date-time. As shown in Fig. 2 (1.1-1.3), once a Yelp URL (e.g., www.yelp.com/biz/brunch-y-ckae-barcelona-2) is selected, the STS recommends MIA as a possible solution for the URL. If a user launches the URL with the MIA, his/her privacy data can be collected by the MIA. Once the MIA collects the data from a user, the MIA then broadcasts the Intent to make the victim app (Yelp) to respond to the URL again.

Link hijacking without STS. We assume that a user intends to open the Yelp app via a browser (e.g., Chrome). As shown in Fig.2 (2.1-2.3), when a user clicks the URL (www.yelp.com/biz/brunch-y-ckae-barcelona-2) to launch the Yelp app, Android ranks all possible apps that can respond to this URL. Android always gives a higher priority to an instant app comparing to a none instant app to respond to a URL. Therefore, the MIA rather than the Yelp app is launched by Android.

Instant app hijacking. If a user intends to launch an instant app, the process can also be hijacked. As shown in Fig. 2 (3.1-3.2), when a user clicks the URL `google.com/tripsapp` to launch the Google Trips instant app; As both our MIA and the victim instant app (Google Trips) are instant apps, Android ranks them based on their package names. In this case, we set a "smaller" package name (`a.example.instantappurlauto`) comparing to the Google Trips (`com.google.android.apps.travel.onthego`). The MIA is then ranked higher than the Google Trips instant app by Android for responding to the URL. As a result, the MIA rather than the Google Trips is launched by Android.

Differences in three attacks. We compare the differences in the three attacks from the following aspects:

•**The victim apps.** As shown in Table 1, only the *instant app hijacking* targets at attacking instant apps. Other attacks only work on typical apps (none instant app);

•**Approaches to launch the attack.** As shown in Fig.2 and Table 1, only *link hijacking with STS* relies on the STS to launch the attack. Other attacks do not rely on the STS.

Remarks. We emphasis that two extra points for these attacks.

• We demonstrate that these attacks can hijack app links (URLs that are mapped to an app). If a URL does not map to any app, it also can be hijacked by *link hijacking with/without STS*.

• The attacks proposed can be launched regardless of the correctness of app link configurations. That is, even if an app link is not correctly configured or the app link verification fails, attacks also can be launched.

Based on the above two points, we can conclude that the attacks proposed are practical with high risks. On the one hand, all URLs can be hijacked by proposed attacks. On the other hand, all apps that define app links can be affected by our attacks.

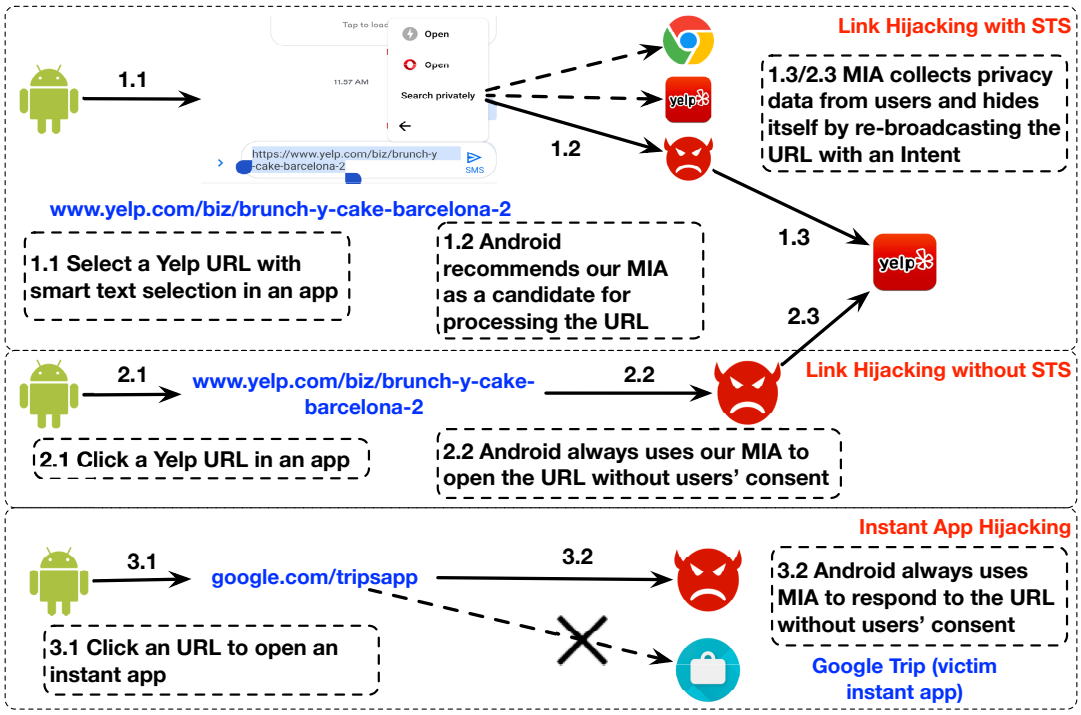

Figure 2: Illustration of three proposed attacks

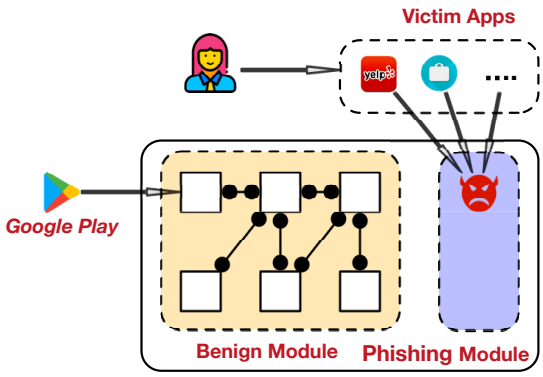

Figure 3: The architecture of the MIA.

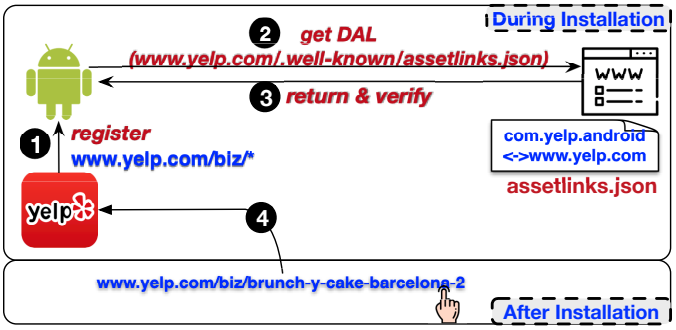

Figure 4: The verification process and usage sample of an app link

3 BACKGROUND

Recall our motivating examples, we highlight that our attacks can be launched no matter the app links are correctly configured or not. Therefore, in this section, we first present the background of app links, then introduce how Android verifies app links, and finally show the cases that can fail the app link verification process.

❶ **App link configuration and release.** To use app links, a developer must register them (in the HTTP(S) scheme) in the app's manifest file. Then, the developer publishes the DAL on the web. For example, in order to use app links in the Yelp app, the Yelp app developers need to register the Yelp domain (`www.yelp.com`) in the app, and also publish the DAL on Yelp's sever at www.yelp.com/.well-known/assetlinks.json.

❷,❸ **Verify app links.** Once a user installs the Yelp app, the verification of app links in the Yelp happens. As shown in Fig. 4, the app registers its app links to Android. To verify these app links, Android extracts the DAL file (assetlinks.json) from the remote server (i.e.,www.yelp.com). As the DAL file defines the authorized app, Android can check whether the package name, scheme, and certificate fingerprint of the app [15] match with a record on the DAL. If and only if the verification of app links passed, the app links become valid.

❹ **Use app links.** Once users click app links, they are redirected to the Yelp app by the Android system.

Invalid app link verification. However, incorrect configuration from either app-end or web-end can fail the link verification process. For the app-end, the incorrect configurations, such as missing the "autoVerify=true" field or invalid the domain names can result in a

failure. For the web-end, the incorrect JSON format, invalid fields in a DAL or invalid namespaces can also be the reasons.

Intent and Intent Filter. Once a user clicks a URL in an app, an `Intent` is sent by the app. In Android, an `Intent` is a messaging object that can be used to request an action from another app component. For example, one `Activity` can start a `Service` (e.g., playing background music) with an `Intent`. There are two types of intents: explicit `Intent` and implicit `Intent`. The explicit `Intent` defines an app to respond to the `Intent` by specifying the app's package name. Whereas, the implicit `Intent` only defines a general action to perform in an `Intent`. For example, an app can use an implicit Intent to request the Google Map app to show a location. Apps that can perform the action defined in an `implicit Intent` get the chance to respond to the `Intent`. To hijack a URL, the MIA must claim that it has the capability to process the link to Android. Only the MIA claims to process a link, Android can consider it as a candidate for the link.

4 ATTACK MODELS

4.1 Overview

In this section, we provide a bird's-eye view of all attack models. We demonstrate an MIA can be exploited as an attack vector for three types of attacks, including link hijacking with STS (§4.2), link hijacking without STS (§4.3), and instant app hijacking attack (§4.4). Even though we launch these attacks with instant apps, the settings of them can be different.

- *Attack vector*: For all attacks, we exploit MIAs as the attack vector.
- *The MIA*: The MIA can either be installed from Google Play or by clicking the the MIA's launching URL in an other app (e.g., Android's SMS app). The reason that attacks can be launched without users' consent is presented in Sec. 2.

4.2 Link Hijacking with Smart Text Selection

The link hijacking with STS aims at preventing users from accessing URLs via STS. When (s)he selects a URL text (e.g. google.com) with STS, Android suggests the MIA for handling the URL. The URL text can come from the app itself or input by users.

The attack steps. In the MIA, for simplicity, we build two `Activities`: `MainActivity` and `LoginActivity`. Recall the architecture of our MIA (see. Fig. 3), the `MainActivity` belongs to the benign module, while the `LoginActivity` belongs to the phishing module. The intent filters associated with them are shown in Fig. 5. The `MainActivity` is the launcher for the MIA and it is bound with `www.<my-own-site>.org/main`. This means when the URL is clicked, the MIA is launched automatically, and Android navigates to the `MainActivity`. The `LoginActivity` is designed for spoofing users.

The `MainActivity` is associated with `<my-own-site>/main`, which is an app link. This is mandatory for launcher `Activity` for instant apps. We also set up a deep link for the malicious `LoginActivity`. The intent filter inside the `LoginActivity` makes the `LoginActivity` can handle the all URLs that match the regular expression
`www.yelp.com/.*`. Note that, as a deep link does not require any verification, the setting for `LoginActivity` does not violate any principles for building an app.

```
<activity android:name=".MainActivity">
    …
    <meta-data
        android:name="default-url"
        android:value="https://www.example.org/main" />
    <intent-filter android:autoVerify="true">
        <action android:name="android.intent.action.VIEW" />
        <category
android:name="android.intent.category.DEFAULT" />
        <category
android:name="android.intent.category.BROWSABLE" />
        <data android:host="www.<my-own-site>.org"
android:pathPattern="/main"
            android:scheme="http" />
    </intent-filter>
</activity>
<activity android:name=".LoginActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category
android:name="android.intent.category.DEFAULT" />
        <data android:host="www.yelp.com"
android:pathPttern="/.*" android:scheme="http"/>
        <data android:host="www.yelp.com"
android:pathPattern="/.*" android:scheme="https"/>
    </intent-filter>
</activity>
```

App Link

for installing & launching the instant app

Deep Link

for hijacking

www.yelp.com/biz/brunch-y-cake-barcelona-2

Figure 5: Link Hijacking Manifest Configuration

The victim apps. This attack targets at apps that access a URL with STS. This attack can work on all kinds of apps, including browsers, in-app browsers (apps that have embedded WebView for opening URLs, e.g., Wechat), and none-browser apps. This attack can be launched on browsers because when a user uses STS for processing text selected, the selection is handled directly by Android rather than the app itself.

The root cause. With deep link defined in Fig. 5, the `LoginActivity` can respond to `www.yelp.com/biz/brunch-y-cake-barcelona-2`. This is because the URL is subject to the regular expression `www.yelp.com/.*`. If the URL is selected, Android looks up for all apps that can respond to the URL. Then, STS suggests all these apps for users to select. Therefore, STS suggests our MIA to users. If users select our MIA, Android uses our MIA (see Fig.6) to respond to the URL. The MIA is thereby launched to respond to the `Intent`.

```
private static List<LabeledIntent> createForUrl(Context context, String
text) {
  if (Uri.parse(text).getScheme() == null) {
    text = "http://" + text;
  }
  return Arrays.asList(new
LabeledIntent(context.getString(com.android.internal.R.string.browse),
context.getString(com.android.internal.R.string.browse_desc), new
Intent(Intent.ACTION_VIEW,
Uri.parse(text)).putExtra(Browser.EXTRA_APPLICATION_ID,
context.getPackageName()),LabeledIntent.DEFAULT_REQUEST_CODE));
}
```

Figure 6: The Smart Text Selection for URL text

4.3 Link Hijacking without Smart Text Selection

The link hijacking aims at preventing users or apps from accessing certain URLs/apps with the MIA. For example, once users click a URL (e.g.,`www.yelp.com/biz/brunch-y-cake-barcelon-2`) in the SMS app, they are redirected to our MIA rather than the Yelp app.

The attack steps. The setting for the MIA follows the same step in §4.2. We also build two `Activities`: `MainActivity` and `LoginActivity`. The intent filters associated with them are shown

in Fig. 5. The only difference is the way of launching this attack. In link hijacking without STS, by clicking a URL, link hijacking occurs. Whereas in link hijacking with STS, users have to select a piece of URL, then STS recommends the MIA to users.

The victim apps. This attack can be launched on apps that access a URL. For example, some apps contain *rate-us* buttons. Once the button is clicked, users are redirected to the MIA. It worth mentioning that this attack cannot work on browsers or in-app browser apps (apps that implement browsers for opening URLs, for example, Wechat). The reason is that browsers or in-app browser apps can handle the URL themselves rather than asking Android to handle the URL.

The root cause. If a URL is clicked, Android looks up all apps that can handle the URL. As our instant app can handle the Intent (see §4.2), Android suggests our MIA to respond to the Intent. Typically, Android asks users to select one app from all candidates to respond to the given link. However, with the instant app, we can successfully escape such prompt. To illustrate this attack, we start from introducing how Android processes a URL click. Once a URL is tapped, the `startActivity` method is called. Android checks whether there is a locally installed app that can resolve this Intent. If multiple apps that can resolve the Intent, Android ranks them based on their package names. Then, Android finds a suitable activity to respond to the Intent with `chooseBestActivity` method (PackageManagerService.java).

Figure 7: The functionality of chooseBestActivity method inside PackageManagerService.java

Inside method `chooseBestActivity`, the system performs the following checking: (1) check whether the first `Activity` has a higher priority (or default). If so, the first `Activity` is used to respond to the Intent; (2) if (1) returns false, Android checks whether there is a preferred app for handling the Intent. If so, the preferred app (a.k.a. saved preference) replies to the Intent; (3) if the checking (2) returns false, Android loops all `Activities` to find whether there is an instant app that can cope with the Intent. If so, the instant app responds to the Intent; (4) if no instant app and no saved preferred app to cope with the Intent, Android shows a list of candidate apps to allow users to select from. The candidate apps are selected based on whether they can handle the Intent. For example, a user can have 5 different apps that can play audios. All these 5 apps are candidates for the task of playing audios (presented as an Intent in a program). From (1) to (4), we can find that if a user does not set any preference, Android places the priority on instant apps. It means in general, an instant app owns a higher priority compared to a none instant app.

Recall our link hijacking scenario, as the `LoginActivity` supports all URLs with the pattern `www.yelp.com/.*`, the MIA can respond to the URL. However, the attack occurs in step (3) and (4). If no app is signed to a higher priority, and no app is set to be the default app, Android checks whether an instant app can cope with the Intent. Consequently, the MIA responds to the Intent. Android provides the MIA a higher priority based on the fact that the app is an instant app. However, the defect is that Android fails to check whether the URL is used for launching the instant app. This attack reveals two defects: first, Android has to verify whether a URL is the one associated with the instant app; and second, Android should not assign a higher priority to instant apps comparing to none instant apps.

The fundamental observations for this vulnerability are: (1) *Android does not verify whether all links claimed in an instant app belong to the same entity (e.g., a developer, a company)*. For an instant app, it is possible to claim both app links and deep links. As deep links are not required to verify, the MIA can claim deep links that it does not own, and (2) *Android authorizes a higher priority to an instant app comparing to an installable app.*

4.4 Instant App Hijacking

The instant app hijacking aims at preventing users from accessing an instant app even if it is installed. Here, we assume that both the MIA and the victim instant app are installed from Google Play.

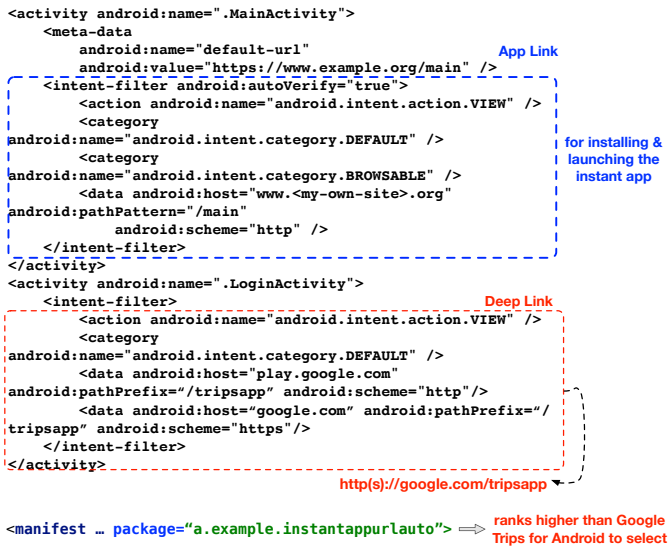

Figure 8: Instant App Hijacking Manifest Configuration

The attack steps. We assume that users download both the victim instant app (e.g., Google Trip instant app `com.google.android.apps.travel.onthego`) and the MIA from Google Play. Once users intend to access the Google Trip instant app, they are redirected to our MIA. As shown in Fig. 8, we set the `MainActivity` same as link hijacking (see Fig. 5) and the malicious `LogActivity` is set to respond the URL `google.com/tripsapp`. This URL is associated with Google Trip instant app (`com.google.android.apps.travel.onthego`). That is, the `LogActivity` is designed for hijacking the Google Trips instant app. The key factor of launching this attack is setting the package name to be one "smaller" than Google Trips' package name based on the dictionary order. Here, we set the package name of our MIA to

`a.example.instantappurlauto` (`a.example.instantappurlauto` < `com.google.android.apps.travel.onthego`).

The victim apps. All instant apps are vulnerable to this attack. If the malicious app is installed prior to the victim instant app, the app user is blocked for downloading and using the victim instant app unless (s)he uninstalls the MIA. If the MIA is installed after the installation of the victim instant app, the app user is blocked for accessing the victim instant app even though the victim instant app is installed on the phone. for both cases, the services from the victim instant app are blocked.

The root cause. By setting the `intent filters` of `LogActivity` as Fig. 8, both the MIA and Google Trip instant app can respond to the URL. If multiple instant apps and apps can resolve one URL, Android ranks them based on package names. Then, Android leverages `chooseBestActivity` method for locating the target app for the `Intent`. Recall the functionality of `chooseBestActivity` introduced in §4.3, if there is no saved preferred app and no app with a higher priority for the given `Intent`, Android checks whether there is an instant app can resolve the `Intent`.

We assign a package name to the MIA with a lower dictionary rank comparing to the victim instant app. As all apps are ranked base on package names, the MIA is ranked higher than the victim app (`a.example.instantappurlauto` < `com.google.android.apps.travel.onthego`). Based on the background presented in §4.3 and Fig. 7, the `chooseBestActivity` is incorrectly chosen the MIA to respond to the URL rather than the victim instant app.

5 DETECTION

In this section, we present our method to detect vulnerable apps that can be attacked.

5.1 Static Analysis

To detect whether an app is vulnerable to proposed attacks, we construct a program dependence graph (PDG) of the app [13]. The PDG consists of the control flow dependencies and the data flow dependencies of the app. To construct a PDG, we collect possible entry points in the app. As Android apps do not specify the entry points (e.g., `main` method for Java application) for execution, we collect the entry points of an app from two parts [21]: (1) lifecycle methods in Android components (e.g., `Activity`). We focus on lifecycle methods of components as they are the standard entry points to the app. Through them, developers can manage the app's components and their behaviors; and (2) UI callbacks. Android allows developers to register UI callbacks for monitoring certain events. For example, the method `onClickListener` is invoked once a button is clicked. In practice, we leverage EdgeMiner [7] to collect all possible entries for a given app. Next, we build PDG of a given app with FlowDroid [3].

Next, we extend the PDG built with FlowDroid to a UI-oriented PDG named UPDG, which models the dependencies and transitions [41, 44] through UI elements (e.g. TextView). A node in UPDG is defined as:

$$n = \{uid, utype, a, c, o\}, \quad (1)$$

in which the *uid* represents the UI element's id. The UI id can be retrieved from the layout files. *utype* represents the type (e.g., TextView) of the UI element. *a* represents the `Activity` context, which means the UI element *uid* is used in the `Activity` *a*. *c* represents the callback method with the element. *o* represents the node in PDG.

To construct the UPDG, we first parse all layout files and the manifest file in the app. From layout files, we extract all UI elements, including text content, name, id (*uid*), and type (*utype*). Then, we match the UI elements with the original PDG nodes. To realize this, we search the statements that are related to UI reference or initialization with `findViewById`. For example, the statement `findViewById(R.id.btn)` can be used to refer to the UI element with id `btn`. Next, we analyze the UI callbacks (e.g., `onClickListener`) that are associated with these UI elements. Then, we retrieve the `Activity` context (*a*) and the corresponding callback (*c*) for a node *n* in UPDG. Last, we append data flows introduced by the inter-component communication (ICC) via implicit Intents [14] to the UPDG. To capture them, we leverage IC3 [33] to locate the source and sink for all Intents in the app. IC3 transforms the ICC problem into a Multi-Valued Composite (MVC) constant propagation problem (i.e., finding all possible values of objects concerned at a particular program point). IC3 specifies the MVC constant propagating problem with the COnstant propAgation (COAL) language and then employs a COAL solver to solve the problem. IC3 infers the arguments in an `Intent` and then finds the target component. We append the data flows introduced by Intents to the UPDG.

5.2 Detecting Link Hijacking with STS

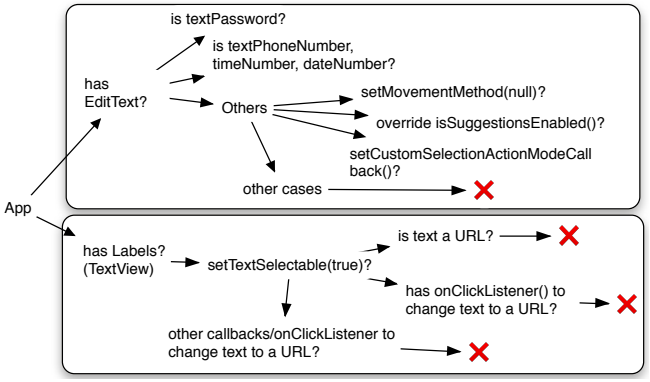

Figure 9: The Workflow for Checking Link Hijacking with STS

In practice, we leverage the workflow defined in Fig. 9 to check whether an app is vulnerable to this attack. To be exact,

• If an app contains `EditTexts` (the text can be edited in an EditText), we check the followings: 1) if the text is a password, phone number, time, or date, the text cannot be visible or cannot be a valid URL. For example, the text in an EditText whose input type is password can not be visible (all characters are presented by "•"); 2) some approaches can be used for preventing users from using STS (e.g., use `setMovementMethod(null)`). These approaches are introduced in §7 in detail. We exclude all these cases as they prohibit users from using STS.

• If an app contains TextViews (a.k.a labels), we check the followings: 1) whether the TextViews can be selected by tracking the `setTextIsSelectable(true)`. If and only if the method

`setTextIsSelectable(true)` is invoked, the text on a TextView can be selected. If the text on the TextView cannot be selected, the attack cannot be launched. It is worth mentioning that by default the text on a TextView cannot be selected; 2) if the content in a TextView is a URL, it can be hijacked with STS; and 3) if the content in a TextView can be changed to a URL, it can be hijacked with STS. For 3), we check all the `onClickListeners` on `TextViews` in the app to evaluate whether the content on `TextViews` can be changed to URLs. Besides, we also check all callbacks and `onClickListeners` in the app to evaluate whether the content on `TextViews` can be changed to URLs.

Implementation. In practice, we transverse the UPDG to find whether any UPDG node that invokes the APIs mentioned above. For a node in the UPDG that is associated with a UI element, we track the node through the UPDG graph to collect all operations performed and allowed on the node. Then, we check whether the node satisfies the conditions in Fig. 9. For example, the expression `writtentext=(EditText)findViewById(R.id.editText1);` defines a EditText object `writtentext`. The `writtentext` is also associated with a UI element whose id is `editText1`. We track the usage of this node (i.e., `writtentext` for simplicity) on the UPDG to find other settings for this object. All settings and operations defined on `writtentext` are then collected. We leverage the workflow in Fig. 9 to evaluate whether attacks can be launched with this node. If so, the app is considered to be vulnerable.

5.3 Detecting Link Hijacking without STS

To determine whether an app is fragile to link hijacking without STS, we use the checking diagram in Fig. 10.

- First, we check whether the app uses any Intent in the code.
- Second, if the app launches a URL with the Intent, we then check whether the app is a browser or contains an in-app browser.
- Third, if the app is not a browser or does not implement any in-app browser, the app is vulnerable to the link hijacking attack without STS.

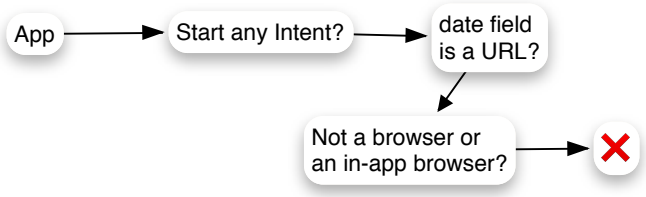

Figure 10: The Workflow for Checking Link Hijacking

As aforementioned, if an app intends to open a URL with an `Intent`, the app is fragile to this attack. However, if the app is a browser or the app contains an in-app browser, the attack cannot be launched. It is because the browser/in-app browser can handle URLs inside itself rather than asking Android to find the targets for the URLs.

Implementation. We first iterate the UPDG to check whether the app launches any `Intent`. First, we track and locate all Intent objects in an app. There are two types of Intents: `explicit` and `implicit` Intent. For explicit Intent, the target of the Intent is defined in the Intent object by specifying the package name. However, to launch a URL, the type of the Intent must be `implicit`. To launch a URL with an Intent, the *data* field is set to the target URL. Therefore, we first locate the Intent objects in the given app. Next, we leverage the taint analysis [3] to track the setting of the *data* field of these Intent objects. If the *data* field is set to a URI (by tracking the type), then we can confirm the app uses an `Intent` to open a URL.

The next step is to detect whether the app is a browser or contains an in-app browser. To do this, we traverse the UPDG to detect whether there is an instance of `WebView` or an instance of type A where A is a subclass of `WebView`. If an app is a browser or implements an in-app browser, the attack cannot be launched.

5.4 Detecting Instant App Hijacking

As presented in Sec. 4.4, the malicious instant app can hijack other instant apps that have *larger* package names in terms of dictionary comparison. Therefore, all instant apps are vulnerable to this attack, as long as the malicious instant app has a deliberately designed package name.

6 EVALUATION

6.1 Evaluation Overview

Apps & DALs To evaluate whether real-world apps are vulnerable to three types of attacks, we crawled 200,000 apps from Google Play and 200,000 apps from Tencent Myapp (the largest third-party app market in China).

To determine whether app links are valid, besides checking the correctness of app links claimed, we also need to check the correctness of the DAL. If and only if both the app itself and the DAL are successfully configured, the app links are valid. To obtain the app links claimed by an app, we plan the following steps:

- **Step 1: Exploring reachable Activities.** We extract all `intent filters` from all `Activities`, whose "categories" contain `BROWSABLE` and `DEFAULT` fields. With these fields, these `Activities` are reachable from a browser;
- **Step 2: Verifying the app link.** For the all `intent filters` extracted from **Step 1**, we extract `intent filters` whose "action" fields contain `VIEW`. It returns `intent filters` either with app links or with customized URLs. Then, we extract `intent filters` with HTTP(s) schemes as they represent the app links;

After **Step 2**, we obtain all app links claimed by the app. For each app link, we extract the domain from the app link. Then, we leverage OpenWPM [12] to visit the domain. We set the loading interval to 15 seconds to allow page loading and URL redirection. Then, we download the corresponding DAL files from remote servers. For a given domain X, its DAL file is located at X/.well-known/asset-links.json.

Evaluation Roadmap. In Research Question (RQ) 1, we discuss the incorrect configurations of app links. Note that, incorrect configurations can make app link verification fails. Apps that fail to pass app link verification are fragile to various attacks [28], such as link hijacking, man-in-the-middle (MITM) attack. They are also vulnerable to attacks proposed in this paper.

In RQ 2 and RQ 3, we evaluate whether real-world apps are robust to link hijacking attacks (with and without STS). In RQ 4, we discuss whether instant apps are robust to instant app hijacking. Last, in RQ 5, we evaluate the accuracy of our MIAFinder tool.

6.2 RQ1: Are real-world apps correctly configure app links?

Motivation. In this RQ, we aim at finding the incorrect configurations of app links. The incorrectness implies that app links are not valid, and it leaves room for attackers.

Methodology. The correctness of app links requires the following checking: 1) the "autoVerify" attribute must be set to be `TRUE` in the app's manifest file. This attribute triggers the verification process of app links declared. Without this attribute, Android does not verify the link; 2) if the domain declared is not valid (i.e., the domain not exists or cannot be visited), the verification of app links cannot be passed; 3) if the domain declared does not contain any DAL, the verification of app links fails as well. For example, if an app claims to associate a domain `www.example.com` and the domain does not publish any DAL, the verification fails. As the `assetlinks.json` must be published under a fixed path (`<domain>/.well-known/assetlinks.json`), if the `assetlinks.json` cannot be found at the path, we consider there is no DAL published; and 4) if the DAL file exists, we check whether the DAL file subjects to its syntax. If and only if the DAL is corrected built, the app links are valid.

Given a DAL, we perform the following steps for evaluating it.

• **Step 1: Checking DAL existences.** We check whether a domain hosts a DAL file. It can be achieved by checking whether the path of the DAL file is reachable. It is possible that a domain does not contain a DAL, and it does not support any app link.

• **Step 2: Verifying the syntax.** As Android forces the syntax of a DAL file, the valid DAL must pass the syntax checking. For example, if a DAL file adds a field that is not supported by the syntax, the verification of the DAL cannot pass.

• **Step 3: Checking fields.** Next, we check the fields and values in the DAL to explore all possible violations in the DAL. For the `relation` field, there are two standard relations (§4). For the `target` field, there are two possible targets, `android_app` and `web`. If a DAL has an invalid field or sets an invalid value, the verification of the DAL fails.

• **Step 4: Include statement checking.** It is also possible to claim statement indirectly by referring an existing DAL with the include statement [17]. Therefore, for this type of DAL, we check the included file with **Step 1** to **Step 3**.

Results. As a result, the incorrect configurations of app end belong to the following categories:

• **Missing "autoVerify=true" field:** To allow Android verifies the app link, the field "autoVerify=true" must be set. Therefore, if an app link lacks such a field, the verification fails.

• **Invalid host format:** The host must be a correct URL in format; otherwise the app link cannot be verified.

• **Inaccessible host:** Even a given host a URL in format, if the URL cannot be accessed, the app link verification can not pass as the DAL cannot be retrieved.

• **Missing fields in Intent-filters:** The valid app link requires the `intent-filter` to specify: a `ACTION_VIEW` action, one or more `data` tags, and two categories (`BROWSABLE` and `DEFAULT`). Missing any field makes the app link invalid.

In addition, we find that the incorrect configurations of DALs belong to the following categories:

• **Incorrect JSON formatting errors:** This type of error is that the DAL is not with a valid JSON format. The errors can be unmatched brackets, unexpected symbols, unknown symbols, and duplicate entities.

• **Incorrect fields:** Some incorrect DALs have field errors, including using undefined fields, missing required fields (e.g.,`namespace`), typo in fields. All these make them invalid.

• **Incorrect Namespace:** The syntax of the DAL requires its namespace to be "android_app" or "web". If a DAL uses invalid namespaces, the app link verification fails.

> **Answer to RQ1:** Among all 200,000 Google Play apps, 8,682 apps use app links. There are only 18.0% of them configure the app links correctly. Among all 200,000 Tencent-Myapp apps, 4,035 apps use app links. There are only 3.1% of them configure the app links correctly.

Implications. This experiment suggests that developers must check the configurations of app links carefully. As for app markets, we suggest app markets for forcing apps to verify domains claimed by apps.

6.3 RQ2: Are real-world apps robust to the general link hijacking attack with STS?

Motivation. In this research question, we intend to evaluate whether real-world apps robust to link hijacking with STS.

Methodology. See Sec. 5.2 for details.

> **Answer to RQ2:** As a result, there are 53,619 Google Play apps (26.8%) that are vulnerable to link hijacking with STS attack. There are 54,650 Tencent-Myapp apps (27.3%) that are vulnerable to this attack.

Implications. For this type of attack, it is the defect introduced by Android. Even if the victim is a browser or contains an in-app browser, the attack can be launched successfully. It is because that STS first obtains the URL selected and then broadcasts an Intent to find the target candidate for the URL. In Sec.7, we proposed four solutions for protecting this attack by either stopping users from selecting any text or customizing the suggestions returned by STS.

6.4 RQ3: Are real-world apps robust to the link hijacking attack without STS?

Motivation. In this research question, we intend to evaluate whether real-world apps are robust to link hijacking without STS. This evaluation aims at finding the risks of this attack.

Methodology. See Section 5.3 for details.

> **Answer to RQ3:** As a result, there are 57,442 Google Play apps (28.7%) that are vulnerable to link hijacking without STS attack. There are 62,496 Tencent-Myapp apps (31.2%) that are vulnerable to this attack.

Implications. As the results show, there is a large number of apps that vulnerable to the link hijacking attack. To prevent from being attacked, an app can implement an in-app browser rather than broadcast the `Intent`. Except for this approach, an app cannot prevent this attack. The reason is that it is Android rather than the victim app to select the target app to respond to the URL. Therefore,

the defect must be fixed from the system layer. Nothing can be done by the victim app to prevent this attack.

6.5 RQ4: Are Instant Apps robust to Instant App Hijacking?

We relax the evaluation of instant app hijacking attack base on the following concerns: first, in §4.4, we illustrate the vulnerability by investigating Android source code. The vulnerability is verified by providing evidence from Android source code; second, the subtlety of this attack is using certain settings in the MIA and exploiting the platform (a.k.a. Android) defects. It means that all instant apps are vulnerable to instant app hijacking.

Though we find the clue that instant apps are not robust to the hijacking from the Android source code, we still test instant app hijacking on all 36 real-world instant apps out of 200,000 apps from Google Play. The instant app hijacking can attack all these instant apps. Note that instant apps can only be published with Google Play rather than other app stores as they require Google Play for installing the victim instant apps [2]. Therefore, we do not consider the apps from Tencent Myapp market for this attack.

Answer to RQ4: All instant apps are vulnerable to instant app hijacking attack.

Implications. This defect implies that Android must carefully select the instant app to respond to a URL if there are more than one instant apps can respond to a link. First, if a link is associated with an instant app, only the instant app that passes the app link verification can respond the link (in our attack, our malicious instant app claims the app via a deep link rather than an app link. Therefore, Google does not force our malicious instant app to verify the link claimed). Second, Android has to ask users to select an instant app to handle if there are multiple instant apps can respond to the URL. Note that, if two instant apps from the same developer/company, it is possible that they can process the same link. Therefore, asking users for their preferences is the solution for this case. Moreover, allowing multiple instant apps for the same URL is an anti-pattern in design.

6.6 RQ5: What is the accuracy of MIAFinder?

Motivation. In this paper, we develop `MIAFinder` to find apps that are vulnerable to proposed attacks. In this RQ, we evaluate the accuracy of the MIAFinder.

Methodology. To evaluate the accuracy of MIAFinder, we randomly select 800 apps from Google Play and then manually checked whether they can be attacked successfully in order to build a benchmark for evaluating our tool. Next, we run the `MIAFinder` to find vulnerable apps. Note that, for an app, if it can be attacked by at least one attack (out of three proposed attacks), we consider it as vulnerable.

Results. As a result, our tool reports 276 vulnerable apps and all of them can be exploited. Therefore, the precision of our tool is 100%.

The recall of our tool is 87%. There are 2 main reasons why our tool misses some vulnerable apps:

• Since the current version of our tool cannot handle native code, it may miss vulnerable apps that define and handle the UI elements through native code. For example, if a vulnerable game app is based on Unity, the UI elements can be defined and handled through C code, and thus our tool misses it;

• Since our tool currently focuses on the default Android UI framework, it misses vulnerable apps that use third-party UI frameworks.

Answer to RQ5: Based on the experiments of 800 apps, we find that the precision of our tool is 100% and the recall is 87%.

7 COUNTERMEASURE

Preventing Link Hijacking with STS. To prevent link hijacking with STS attack, we propose 4 solutions.

• **Solution 1.** If a developer plans to use TextView (a.k.a, label) in an app, (s)he can use the `setTextSelecttable(false)` to prevent users from selecting any text in a label;

• **Solution 2.** If a developer plans to use EditText (users can edit text in EditText), (s)he can use `setMovementMethod(null)` to prevent users from selecting any text in an EditText.

• **Solution 3.** Developers can override the `isSuggestionsEnabled()` method in default TextView or EditText. The `isSuggestionsEnabled()` returns a Boolean value to indicate whether or not suggestions are enabled on this TextView or EditText.

• **Solution 4.** Another solution is using the `setCustomSelectionActionModeCallback` API. This API allows developers to customize the popup menu if a piece of text is selected.

All these approaches aim at preventing users from selecting any text in UIs or customizing the popup menu for STS. Once users cannot select any text in UIs, the STS cannot work.

Preventing Link Hijacking without STS. To address this, we propose a novel API named `verifyDomainPackage`. As shown in Fig. 11, Android can invoke `verifyDomainPackage`, whose arguments are the received an `Intent` and an `Activity`. The `Intent` contains the target URL to open. The `Activity` represents a candidate component to respond to the `Intent`. First, it retrieves the package name of the candidate app (e.g., the MIA) that can respond to the `Intent` (Fig. 11: 1.1). Second, it extracts the candidate app's signature stored in Android (Fig. 11: 1.2-1.3). Third, it downloads the DAL for the URL given in the `Intent` (Fig. 11: 1.4-1.5). Last, it exams whether the candidate app can respond to the URL by checking the package name, signature with the DAL downloaded (Fig. 11: 1.6). The `verifyDomainPackage` can be used to check whether there is an MIA that intends to hijack URLs.

With the `verifyDomainPackage` API, Android can check whether a candidate app (i.e., `Activity`) can be used to respond to a URL. It worth mentioning that using `verifyDomainPackage` can prevent the link hijacking attack with STS.

Preventing Instant App Hijacking Attack. To prevent the instant app hijacking attack, Android can scan all instant apps installed. If Android finds an instant app that claims a URL that is not owned by the instant app, Android must inform app users about this case. To check whether an instant app owns all URLs claimed, Android can use the `verifyDomainPackage` API. If an instant app claims a URL that it does not own, the instant app can be a malicious one. Android can notify users of the potential risk of this MIA.

To wrap up, the novel API `verifyDomainPackage` proposed can be leveraged to prevent all three attacks aforementioned. We also

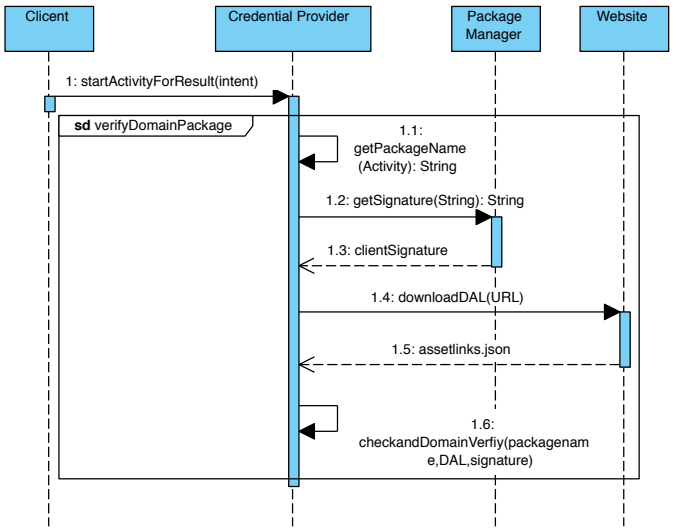

Figure 11: The Workflow for `verifyDomainPackage` API

build our patch and verify it with the `verifyDomainPackage` API on Android 10.0.

8 RELATED WORK

Deep Links. Ma et al. [31]'s work Aladdin helps developers automatically release deep links for an app. Aladdin first computes the paths to reach each Activity and Fragment in a given app. Then, it construct a *proxy* Activity to bind all deep links to the *proxy* Activity. The *proxy* Activity is also in charge of managing all deep links. Hu et al. [22] proposed a framework called Elix, which aims at extracting all valid deep links that are defined in an app. Elix extracts app links with a path-selective taint analysis. It leverages a taint analysis that starts from the `Activity.getIntent()` for taint analysis and prunes infeasible paths with the symbolic execution. Liu et al. [28] conduct an empirical measurement on various mobile deep links across apps and websites to explore the incorrect configurations for deep links. Different from all these works, our work aims at revealing the defects in app links rather than leveraging (or constructing) app links (or deep links).

Instant App. The only work that related to instant apps is proposed by Aonzo et al. [1]. In [1], Aonzo et al. reported the design defects in password manager apps and mentioned the misuse of accessibility service can result in security problems. Even though work [1] leverages instant app as an attack vector, our work has different research targets and unique contributions: [1] aims at using instant apps for phishing rather than uncovering defects in the app link schemes. Whereas, our work targets at exploring the defects rooted in Android, including link verification, access control, and priority ranking.

App Browser Security. Some works aim at attacking mobile browsers and in-app browsers [10, 19, 30, 32, 37, 38]. Chin et al. [10] reveal two WebView vulnerabilities, including *excess authorization* and *file-based cross-zone scripting*. Tuncay et al. [37] present that Draco, which enables developers to specify a set of policies to only allow desired access. Wang et al. [38] reveal the cross-origin risk in Android and iOS browsers and in-app browser apps. With the cross-origin attack, malicious apps can obtain a mobile user's authentication credentials and record users' behavior. Different from all these works, our work targets at measuring the vulnerability of app links in terms of using an instant app as an attack vector.

App-to-app Communication. The works on the app to app communication leverage both static and dynamic analysis. On the one hand, researchers leverage static analysis to detect privacy leakage from the victim app to the malicious app [5, 6, 18, 25–27, 29, 35, 42, 43] using the call graph and taint analysis [23, 34]. On the other hand, some works explore privacy the leakage issues with dynamic analysis [11, 20, 24, 40]. Different from all these works, our work highlights the deep link vulnerability introduced with instant apps.

9 CONCLUSION

In this paper, we revisit app links defined by Android and reveal three attack models that can be exploited. Our attacks showed strong evidence that existing limitations in verifying instant apps and drawbacks in launching instant apps (e.g., an instant app should not be given a higher priority comparing to a typical app). To evaluate whether existing Android apps are fragile to these attacks, we conduct a large-scale empirical study on 200,000 Android apps on Google Play and 200,000 apps on Tencent Myapp. As a result, there are 53,619 Google Play apps and 54,650 Tencent-Myapp apps that vulnerable to link hijacking with smart text selection; 57,442 Google Play apps and 62,496 Tencent-Myapp apps that vulnerable to link hijacking without smart text selection; and all instant apps are vulnerable to instant app hijacking. Finally, we make a series of suggestions to countermeasure the attacks we proposed.

10 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their helpful comments. This research is partially supported by ShanghaiTech University Start-up Research Fund, the Hong Kong General Research Fund (No. 152223/17E, 152239/18E), Australian Research Grants (No. DP200101328), the National Natural Science Foundation of China (No. 61702045) and China Postdoctoral Science Foundation (No.2020M673137).

REFERENCES

[1] Simone Aonzo, Alessio Merlo, Giulio Tavella, and Yanick Fratantonio. 2018. Phishing Attacks on Modern Android. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1788–1801.

[2] Instant App. 2019. https://developer.android.com/topic/google-play-instant

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (2014), 259–269.

[4] Tanzirul Azim, Oriana Riva, and Suman Nath. 2016. uLink: Enabling user-defined deep linking to app content. In *14th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[5] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. 2015. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Transactions on Software Engineering (TSE)* 41, 9 (2015), 866–886.

[6] Amiangshu Bosu, Fang Liu, Danfeng Yao, and Gang Wang. 2017. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*. 71–85.

[7] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[8] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *23rd USENIX Security Symposium (Security)*. 1037–1052.

[9] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 239–252.

[10] Erika Chin and David Wagner. 2014. Bifocals: Analyzing WebView Vulnerabilities in Android Applications. In *Revised Selected Papers of the 14th International Workshop on Information Security Applications*. 138–159.

[11] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2 (2014), 5:1–5:29.

[12] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1388–1401.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (1987), 319–349.

[14] Google. 2019. https://developer.android.com/reference/android/content/Intent

[15] Google. 2019. Android App Signing. https://developer.android.com/studio/publish/app-signing

[16] Google. 2019. Deep Link. https://developer.android.com/training/app-links/deep-linking

[17] Google. 2019. Digital Asset Links. https://developers.google.com/digital-asset-links/

[18] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*. 1–16.

[19] Behnaz Hassanshahi, Yaoqi Jia, Roland H. C. Yap, Prateek Saxena, and Zhenkai Liang. 2015. Web-to-Application Injection Attacks on Android: Characterization and Detection. In *European Symposium on Research in Computer Security*. 577–598.

[20] Roee Hay, Omer Tripp, and Marco Pistoia. 2015. Dynamic Detection of Inter-application Communication Vulnerabilities in Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. 118–128.

[21] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y. Ko, and Lukasz Ziarek. 2013. Flow Permissions for Android. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 652–657.

[22] Yongjian Hu, Oriana Riva, Suman Nath, and Iulian Neamtiu. 2019. Elix: Path-Selective Taint Analysis for Extracting Mobile App Links. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 193–206.

[23] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. 106–117.

[24] Yiming Jing, Gail-Joon Ahn, Adam Doupé, and Jeong Hyun Yi. 2016. Checking Intent-based Communication in Android with Intent Space Analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*. 735–746.

[25] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP)*. 1–6.

[26] L. Li, A. Bartel, T. F. BissyandÃĺ, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 280–291.

[27] F. Liu, H. Cai, G. Wang, D. Yao, K. O. Elish, and B. G. Ryder. 2017. MR-Droid: A Scalable and Prioritized Analysis of Inter-App Communication Risks. In *2017 IEEE Security and Privacy Workshops (S & P)*. 189–198.

[28] Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, and Gang Wang. 2017. Measuring the Insecurity of Mobile Deep Links of Android. In *26th USENIX Security Symposium (Security)*. 953–969.

[29] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. 229–240.

[30] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. 2011. Attacks on WebView in the Android System. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*. 343–352.

[31] Yun Ma, Ziniu Hu, Yunxin Liu, Tao Xie, and Xuanzhe Liu. 2018. Aladdin: Automating Release of Deep-Link APIs on Android. In *Proceedings of the 2018 World Wide Web Conference (WWW)*. 1469–1478.

[32] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. 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 (POPL)*. 469–484.

[33] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. 77–88.

[34] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 331–341.

[35] D. Sbirlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. 2013. Automatic detection of inter-application permission leaks in Android applications. *IBM Journal of Research and Development* 57, 6 (2013), 10:1–10:12.

[36] Smart Text Selection. 2019. https://www.android.com/versions/oreo-8-0/

[37] Guliz Seray Tuncay, Soteris Demetriou, and Carl A. Gunter. 2016. Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 104–115.

[38] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 635–646.

[39] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. 2015. Cracking App Isolation on Apple: Unauthorized Cross-App Resource Access on MAC OS X and IOS. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 31âĂŞ43.

[40] Kun Yang, Jianwei Zhuge, Yongke Wang, Lujue Zhou, and Haixin Duan. 2014. IntentFuzzer: Detecting Capability Leaks of Android Applications. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*. 531–536.

[41] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 658–668.

[42] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security (CCS)*. 1043–1054.

[43] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. 2018. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1165–1178.

[44] L. L. Zhang, C. M. Liang, Z. L. Li, Y. Liu, F. Zhao, and E. Chen. 2018. Characterizing Privacy Risks of Mobile Apps with Sensitivity Analysis. *IEEE Transactions on Mobile Computing (TMC)* 17, 2 (2018), 279–292.