Measuring the Insecurity of Mobile Deep Links of Android

Fang Liu, Chun Wang, Andres Pico, Danfeng Yao, Gang Wang
Department of Computer Science, Virginia Tech
{*fbeyond, wchun, andres, danfeng, gangwang*}*@vt.edu*

Abstract

Mobile deep links are URIs that point to specific locations within apps, which are instrumental to web-to-app communications. Existing "scheme URLs" are known to have hijacking vulnerabilities where one app can freely register another app's schemes to hijack the communication. Recently, Android introduced two new methods "App links" and "Intent URLs" which were designed with security features, to replace scheme URLs. While the new mechanisms are secure in theory, little is known about how effective they are in practice.

In this paper, we conduct the first empirical measurement on various mobile deep links across apps and websites. Our analysis is based on the deep links extracted from two snapshots of 160,000+ top Android apps from Google Play (2014 and 2016), and 1 million webpages from Alexa top domains. We find that the new linking methods (particularly App links) not only failed to deliver the security benefits as designed, but significantly worsen the situation. First, App links apply link verification to prevent hijacking. However, only 194 apps (2.2% out of 8,878 apps with App links) can pass the verification due to incorrect (or no) implementations. Second, we identify a new vulnerability in App link's preference setting, which allows a malicious app to intercept arbitrary HTTPS URLs in the browser without raising any alerts. Third, we identify more hijacking cases on App links than existing scheme URLs among both apps and websites. Many of them are targeting popular sites such as online social networks. Finally, Intent URLs have little impact in mitigating hijacking risks due to a low adoption rate on the web.

1 Introduction

With the wide adoption of smartphones, mobile websites and native apps have become the two primary interfaces to access online content [10, 44]. Today, a user can easily launch apps from websites with preloaded *context*, which becomes instrumental to many key user experiences. For instance, from a restaurant's home page, users can tap a hyperlink to launch the phone app and call the restaurant, or launch Google Maps for navigation. Recently, users can even search in-app content with a web-based search engine (*e.g.*, Google) and directly launch the target app by clicking the search result [5].

The key enabler of web-to-mobile communication is mobile deep links. Like web URLs, mobile deep links are universal resource identifiers (URI) for content and functions within apps [49]. The most widely used deep link is *scheme URL* supported by both Android [7] and iOS [3] since 2008. If an app wants to be launched from the web, the app can register URI schemes to the mobile OS during installation. For example, the Facebook app registers "`fb://profile`" to open user profiles. Later when the link "`fb://profile/user1`" is clicked on the web, OS then can direct users to the Facebook app.

Threats to Mobile Deep Links. Despite the convenience, researchers have identified serious security vulnerabilities in scheme URLs [18, 19, 55]. The most significant one is *link hijacking*, where one app can register another app's scheme and induce the mobile OS to open the wrong app. Fundamentally, link hijacking is possible because there is no restriction on what schemes apps can register. A malicious app may register "`fb`" to hijack the deep link request to the Facebook app to launch itself. This allows the malicious apps to perform phishing attacks (*e.g.*, displaying a fake Facebook login box) or steal sensitive data carried by the link (*e.g.*, PII) [19, 35]. Even though Android and iOS may prompt users before launching an app, there are many cases where such prompting is skipped without user knowledge.

Recently, two new deep link mechanisms were proposed to address the security risks in scheme URLs: App link and Intent URL. 1) *App Link* [6, 9] was introduced to Android and iOS in 2015. It no longer al-

lows developers to customize schemes, but exclusively uses HTTP/HTTPS scheme. To prevent hijacking, App links introduced a way to verify the app-to-link association. More specifically, mobile OS verifies a registered link (*e.g.*, `https://facebook.com/profile`) by contacting the corresponding web host (`facebook.com`) for verification. This prevents apps other than Facebook to claim this link. 2) *Intent URL* [2] is another solution introduced in 2013, which only works on Android. Intent URL defines how deep links should be called by websites. Instead of calling "`fb://profile`", Intent URL explicitly specifies the destination app identifier (*i.e.*, package name) in the parameter to avoid confusion.

Measurements. While most existing works focus on vulnerabilities in scheme URLs [18, 19, 55], little is known about how widely App links and Intent URLs are adopted, and how effective they are in mitigating the threat in practice. In this paper, we conduct the first large-scale measurement on the current ecosystem of mobile deep links. Our goal is to detect and measure link hijacking vulnerabilities across the web and mobile apps, and understand the effectiveness of new linking mechanisms in battling hijacking attacks.

We perform extensive measurements on a large collection of mobile apps and websites. To measure the adoption of different mobile deep links, we collected two snapshots of 160,000+ most popular Android apps from Google Play in 2014 and 2016, and crawled 1 million web pages (using a dynamic crawler) from Alexa top domains. We primarily focus on Android for its significant market share (87%) [29] and availability of apps. We also perform a subset of analysis on iOS deep links. At the high-level, our method is to extract the link registration entries (URIs) from apps, and then measure their empirical usage on websites. To detect hijacking attacks, we group apps that register the same URIs as link collision groups. We find that not all link collisions are malicious — certain links are expected to be shared such as links for common functionality (*e.g.*, "`tel`") or third-party libraries (*e.g.*, "`zxing`"). We develop methods to identify malicious hijacking attempts.

Findings. Our study has four surprising findings, which lead to one overall conclusion: the newly introduced deep link solutions not only fail to improve security, but significantly increase hijacking risks for users.

First, App links' verification mechanism fails in practice. Surprisingly, among 8,878 Android apps with App links, only 194 (2.2%) correctly implement link verification. The reasons are a combination of the lack of motivation from app developers and various developer mistakes. We confirm a subset of mistakes in iOS App links too: 1,925 out of 12,570 (15%) fail the verification due to server misconfigurations, including popular apps such as Airbnb.

Second, we uncover a new vulnerability in App links, which allows malicious apps to *stealthily* intercept HTTP/HTTPS URLs in the browser. The root cause is that Android grants excessive permissions to unverified App links through the preference setting. For an unverified App link, Android by default will prompt users to choose between the app and the browser. To disable promoting, users may set a "preference" to always use the app for this link. This preference is overly permissive, since it not only disables prompting for the current link, but all other unverified links registered by the app. A malicious app, once received preference, can hijack any sensitive HTTP/HTTPS URLs (*e.g.*, to a bank website) without alerting users. We validate this vulnerability in the latest Android 7.1.1.

Third, We detect more malicious hijacking attacks on App links (1,593 apps) than scheme URLs (893 apps). Case studies show that popular websites (*e.g.*, "`google.com`") and apps (*e.g.*, Facebook) are common targets for traffic hijacking. In addition, we identify suspicious apps that act as the man-in-the-middle between websites and the original app to record sensitive URLs and the parameters (*e.g.*, "`https://paypal.com`").

Finally, Intent URLs have very limited impact in mitigating hijacking risks due to the low adoption rate among websites. Only 452 websites out of the Alexa top 1 million contain Intent URLs (0.05%), which is a much lower ratio than that of App links (48.0%) and scheme URLs (19.7%). Meanwhile, among these websites, App links drastically increase the number of links that have hijacking risks compared to existing vulnerable scheme URLs

To the best of our knowledge, our study is the first empirical measurement on the ecosystem of mobile deep links across web and apps. We find the new linking methods not only fail to deliver the security benefits as designed, but significantly worsen the situation. There is a clear mismatch between the security design and practical implementations due to the lack of incentives of developers, developer mistakes, and inherent vulnerabilities in the link mechanism. Moving forward, we propose a list of suggestions to mitigate the threat. We have reported the over-permission vulnerability to the Google Android team. The detailed plan for further notification and risk mitigation is described in §8.

2 Background and Research Goals

Mobile deep links are URIs that point to specific locations within mobile apps. Through deep links, websites can initiate useful interactions with apps, which is instrumental to many key user experiences, for example, opening apps, sharing and bookmarking in-app pages [49],

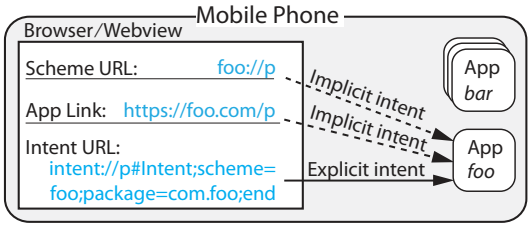


Figure 1: Three types of mobile deep links: Scheme URL, App Link and Intent URL.

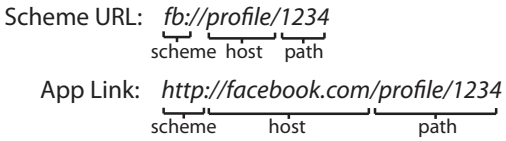


Figure 2: URI syntax for Scheme URLs and App links.

and searching in-app content using search engines [5]. In the following, we briefly introduce how deep links work and the related security vulnerabilities. Then we describe our research goals and methodology.

2.1 Mobile Deep Links

To understand how deep links work, we first introduce inter-app communications on Android. An Android app is essentially a package of software *components*. One app's components can communicate with another app's components through *Intent*, a messaging object characterized "action", "category" and "data". By sending an intent, one app can communicate with the other app's front-end *Activities*, or background *Services*, *Content Providers* and *Broadcast Receivers*.

Mobile deep links trigger a particular type of intent to enable communications between the web and mobile apps. As shown in Figure 1, after users click on a link in the browser (or in-app WebView), the browser sends an intent to invoke the corresponding component in the target app. Unlike app-to-app communication, mobile deep link can only launch front-end Activity in the app.

Mobile deep links work in two simple steps: 1) Registration: an app "`foo`" should first register its URIs ("`foo://`" or "`https://foo.com`") to the mobile OS during installation. The URIs are declared in the in the "data" field of *intent filters*. 2) Addressing: when "`foo://`" is clicked, mobile OS will search all the intent filters for a potential match. Since the link matches the URI of app "`foo`", mobile OS will launch this app.

2.2 Security Risks of Deep Linking

Hijacking Risk in Scheme URL. Scheme URL is the first generation of mobile deep links, and is the least secure one. It was introduced since Android 1.0 [7] and iOS 2.0 [3] in 2008. Figure 2 shows the syntax of a scheme URL. App developers can customize any schemes and URIs for their app without any restriction.

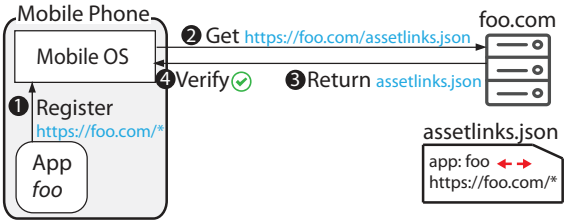


Figure 3: App link verification process.

Prior research has pointed out key security risks in scheme URLs [19, 55], given that any app can register other apps' schemes. For example, apps other than Facebook can also register "`fb://`". When a deep link is clicked, it triggers an "implicit intent" to open any app with a matched URI. This allows a malicious app to hijack the request to the Facebook app to launch itself, either for phishing (*e.g.*, displaying a fake Facebook login box), or stealing sensitive data in the request [19, 35].

With an awareness of this risk, Android lets users be the security guard. When multiple apps declare the same URI, users will be prompted (with a dialog box) to select/confirm their intended app. However, if the malicious app is installed but the victim app is not, the malicious app will automatically skip the prompting and hijack the link without user knowledge. Even when both apps are installed, the malicious app may trick users to set itself as the "preference" and disable prompting. Historically speaking, relying on end-users as the sole security defense is risky since users often fail to perceive the nature of an attack, leading to bad decisions [12, 22, 53].

Solution1: App Link. App Link was introduced recently in October 2015 to Android 6.0 [6] as a more secure version of deep links. It was designed to prevent hijacking with two mechanisms. First, the authentic app can build an association with the corresponding website, which allows the mobile OS to open the App link exclusively using the authentic app. Second, App link no longer allows developers to customize their own schemes, but exclusively uses the `http` or `https` scheme.

Figure 3 shows the App link association process. Suppose app "*foo*" wants to register "`http://foo.com/*`". Mobile OS will contact the server at "`foo.com`" for verification. The app's developer needs to set up an association file "assetlinks.json" beforehand under the root directory ("/.well-known/") of the `foo.com` server. This file must be hosted on an HTTPS server. If the file contains an entry that certifies that app "*foo*" is associated with the link "`http://foo.com/*`", the mobile OS will confirm the association. The association file

contains a field called "`sha256_cert_fingerprints`", which is the SHA256 fingerprint of the associated app's signing certificate. The mobile OS is able to verify the fingerprint and prevent hijacking because only the authentic app has the corresponding signing certificate. Suppose a malicious app "`bar`" also wants to register "`http://foo.com/*`", the verification will fail, assuming the attacker cannot access the root of `foo.com` server to modify the association file and the fingerprint.

The iOS version of App links is called universal link, introduced at iOS 9.0 [9], which has the same verification process. The association file for iOS is "apple-app-site-association". However, iOS and Android have different policies to handle *failed verifications*. iOS prohibits opening unverified universal links in apps. Android, however, leaves the decision to users: if an unverified link is clicked, Android prompts users to choose if they want to open the link in the app or the browser.

Solution 2: Intent URL. Intent URL was introduced in 2013 and only works on Android [2]. Intent URLs prevent hijacking by changing how the deep link is called on the website. As shown in Figure 1, instead of calling "`foo://p`", Intent URL is structured as "`intent://p/#Intent;scheme=foo;package=com.foo;end`" where the package name of the target app is explicitly specified. Package name is a unique identifier for an Android app. Clicking an intent URL will launch an "explicit intent" to open the specified app.

Compared to scheme URLs and App links, Intent URL does not need special URI registration on the app. Intent URL can invoke the same interfaces defined by the URIs of scheme URLs or App links, as well as other exposed components [2].

2.3 Research Questions

While the hijacking risk of scheme URLs has been reported by existing research [18, 19, 55], little is known about how prevalently this risk exists among apps, and how effective the new mechanisms (App links and Intent URLs) are in reducing this risk in practice. We hypothesize that upgrading from scheme URL to App link/Intent URL is a non-trivial task, considering that scheme URLs may already have significant footprints on the web. Mobile platforms might be able to enforce changes to apps through OS updates, but their influence on the web is likely less significant. In this paper, we conduct the first large-scale measurement on the mobile deep link ecosystem to understand the adoption of different linking methods and their effectiveness in battling hijacking threats.

Threat Model. Our study focuses on *link hijacking threat* since this is the security issue that App Links and Intent URLs aim to address. Link hijacking happens when a malicious app registers the URI that belongs to the victim app. If mobile OS redirects the user to the malicious app, it can lead to phishing (*e.g.*, the malicious app displays forged UI to lure user passwords) or data leakage (*e.g.*, the deep link may carry sensitive data in the URL parameters such as PII and session IDs) [19, 35]. In this threat model, mobile OS and browser (or WebView) are not the targets of the attack, and we assume they are not malicious.

Link Type	Conditions			Prompt User?
	>1 Apps	Set As Preference	Link Verified	
Scheme URL	✓	✗	/	✓
	✓	✓	/	✗
	✗	✗	/	✗
	✗	✓	/	✗
App Link*	/	✗	✗	✓
	/	✓	✗	✗
	/	✗	✓	✗
	/	✓	✓	✗
Intent URL	/	/	/	✗

Table 1: Conditions for whether users will be prompted after clicking a deep link on Android. *App Links always have at least one matched app, the mobile browser.

The Role of Users. Users also play a role in this threat model. After clicking on a deep link, a user may be prompted with a dialog box to confirm the destination app. As shown in Table 1, prompting can be skipped in many cases. For *scheme URLs*, a malicious app can skip prompting if the victim app is not installed, or by tricking users to set the malicious app as the "preference". *App link* can skip prompting if the link has been verified. Otherwise, users will be prompted to choose between the browser and the app. *Intent URLs* will not prompt users at all since the target app is explicitly specified.

Our Goals. Our study seeks to answer key questions regarding how mobile deep links are implemented in the wild and their security impact. We ask three sets of questions. *First*, how prevalently are different deep links adopted among apps over time? Are App links and Intent URLs implemented properly as designed? *Second*, how many apps are still vulnerable to hijacking attacks? How many vulnerable apps are exploited by other real-world apps? *Third*, how widely are hijacked links distributed among websites? How much do App links and Intent URLs contribute to mitigating such links?

To answer these questions, we first describe data collection (§3), and measure the adoption of App links and scheme URLs among apps (§4). We perform extensive security analyses to understand how effective App links can prevent hijacking (§5), and then describe the method to detect hijacking attacks among apps (§6). Finally, we move to the web to measure the usage of Intent URLs,

and the prevalence of hijacked links (§7). In §8, we summarize key implications and discuss possible solutions.

3 Datasets

We collected data from both mobile apps and websites, including two snapshots of 160,000+ most popular Android apps in 2014 and 2016, and web pages from Alexa top 1 million domains.

Mobile Apps. To examine deep link registration, we crawled two snapshots of mobile apps from Google Play. The first snapshot *App2014* contains 164,322 most popular free apps from 25 categories in December 2014 (crawled with an Android 4.0.1 client). In August 2016, we crawled a second snapshot of top 160,000 free apps using an Android 6.0.1 client. We find that 48,923 apps in App2014 are no longer listed on the market in 2016. 4,963 apps in 2014 snapshot fell out of the top 160K list in 2016. To match the two datasets, we also crawled these 4,963 apps in 2016, forming an *App2016* dataset of 164,963 apps. The two snapshots have 115,399 overlapping apps. For each app in *App2016*, we also obtained the developer information, downloading count, review count and rating.

Our app dataset is biased towards popular apps among the 2.2 million apps in Google Play [48]. Since these popular apps have more downloads, potential vulnerabilities could affect more users. Our result can serve as a lower bound of empirical risks.

Alexa Top 1 Million Websites. To understand deep link usage on the web, we crawled Alexa top 1 million domains [1] in October 2016. We simulate using an Android browser (Android 6.0.1, Chrome/41/0/2272.96) to visit these web domains and load both static HTML page (index page) and the dynamic content from JavaScript. This is done using modified OpenWPM [25], a headless browser-based crawler. For each visit, the crawler loads the web page and waits for 300 seconds allowing the page to load the dynamic content, or perform the redirection. We store the final URL and HTML content. This crawling is also biased towards popular websites, assuming that deep links on these sites are more likely to be encountered by users. We refer this dataset as *Alexa1M*.

4 Deep Link Registration by Apps

In this section, we start by analyzing mobile apps to understand deep link registration and adoption. In order to receive deep link requests, an app needs to register its URIs to mobile OS during installation. Our analysis in this section focuses on Scheme URLs and App links. For Intent URLs, as described in §2, developers do not need special registrations in the app. Instead, it is up to the *websites* to decide whether to use Intent URLs or scheme URLs to launch the app. We will examine the adoption Intent URLs later by analyzing web pages (§7).

We provide an overview of deep link adoption by analyzing 1) how widely the scheme URLs are adopted among apps, and 2) whether App links are in the process of replacing scheme URLs for better security.

4.1 Extracting URI Registration Entries

Android apps register their URIs in the manifest file (`AndroidManifest.xml`). Both Scheme URLs and App Links are declared in Intent filters as a set of matching rules, which can either be actual links (`fb://login/`) or a wild card (`fb://profile/*`). Since there is no way to exhaustively obtain all links behind a wild card, we treat each matching rule as a registration entry. Given a manifest file, we extract deep link entries in three steps:

- **Step1: Detecting Open Interfaces.** We capture all the Activity intent filters whose "category" field contains both *BROWSABLE* and *DEFAULT*. This returns all the components that are reachable from the web.
- **Step2: Extracting App Link.** Among intent filters in Step 1, we capture those whose "action" contains *VIEW*. This returns intent filters with either App Links or Scheme URLs in their "data" fields[1]. We extract App Link URIs as those with http/https scheme. Note that App Link intent filters have a special field called *autoVerify*. If its value is TRUE, then mobile OS will perform verification on the App link.
- **Step3: Extracting Scheme URL.** All the non-http/https URIs from Step2 are Scheme URLs.

We apply the above method to our dataset and the result is summarized in Table 2. Among the 160K apps in *App2016*, we find that 20.3K apps adopt scheme URLs and 8.9K apps adopt App links. Note that for the apps in *App2014* (Android 4.0 or lower), App Link had not been introduced to Android yet. We find that 4,545 apps in *App2014* register http/https URIs, which are essentially scheme URLs with "`http`" or "`https`" as the scheme. For consistency, we still call these http/https links as App links, but link verification is not supported for these apps.

4.2 Scheme URL vs. App Link

Next, we compare the adoption of Scheme URLs and App links across time, app categories and app popularity. We seek to understand if the new App links are on the way of replacing Scheme URLs.

[1]The rest intent filters whose "action" is not *VIEW* can still be triggered by Intent URLs.

Dataset	Total Apps	Apps accept Scheme URLs	Apps accept App Links	Apps accept either Links	Unique Schemes	Unique Web Hosts
App2014	164,322	10,565 (6.4%)	4,545 (2.8%)	12,428 (7.6%)	8,845	6,471
App2016	164,963	20,257 (12.3%)	8,878 (5.4%)	23,830 (14.5%)	18,839	18,561

Table 2: Two snapshots of Android apps collected in 2014 and 2016. 115,399 apps appear in the both datasets; 48,923 apps in App2014 are no longer listed on the market in 2016; App2016 has 49,564 new apps.

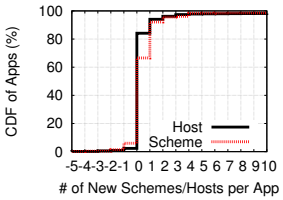


Figure 4: # of new schemes and app link hosts per app between 2014 and 2016.

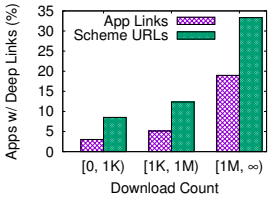


Figure 5: % of apps w/deep links; apps are divided by download count.

Adoption over Time. As shown in Table 2, there are significantly more apps that started to adopt deep links from 2014 to 2016 (about 100% growth). However, the growth rates are almost the same for App links and Scheme URLs. There are still 2-3 times more apps using scheme URLs than those with App links. Apps links are far from replacing scheme URLs.

Figure 4 specifically looks at apps in both snapshots. We select those that adopt either type of deep links in either snapshot (13,538 apps), and compute the differences in their number of schemes/hosts between 2014 and 2016. We find that the majority of apps (over 96.2%) either added more deep links or remained the same. Almost no apps removed or replaced scheme URLs with App links. The conclusion is the same when we compare the number of URI rules (omitted for brevity). This suggests that scheme URLs are still heavily used, exposing users to potential hijacking threat.

App Popularity. We find that deep links are more commonly used by popular apps (based on download count). In Figure 5, we divide apps in 2016 into three buckets based on their download count: $[0, 1K)$, $[1K, 1M)$, $[1M, \infty)$. Each has 20,654, 127,323 and 5,223 apps respectively. Then we calculate the percentage of apps that adopt deep links in each bucket. We observe that 33% of the 5,223 most popular apps adopt scheme URL, and the adoption rate goes down to 8% for apps with <1K downloads. The trend is similar for App links. In addition, we find that apps *with* deep links have averagely 4 million downloads per app, which is orders of magnitude higher than apps *without* deep links (125K downloads per app). As deep links are associated with popular apps, potential vulnerabilities can affect many users.

App Categories. Among the 25 app categories, we find that the following categories have the highest deep link adoption rate: SHOPPING (25.5%), SOCIAL (23.4%), LIFESTYLE (21.0%), NEWS_AND_MAGAZINES (20.5%) and TRAVEL_AND_LOCAL (20.2%). These apps are content-heavy and often handle user personally identifiable information (*e.g.*, social network app) and financial data (*e.g.*, shopping app). Link hijacking targeting these apps could have practical consequences.

5 Security Analysis of App Links

Our result shows that App links are still not as popular as scheme URLs. Then for apps that adopt App links, are they truly secure against link hijacking? As we discussed in §2.2, App link was designed to prevent hijacking through a link verification process. If a user clicks on an unverified App link, the mobile OS will prompt the user to choose whether he/she would like to open the link in the browser or using the app. In the following, we empirically analyze the security properties of App links in two aspects. First, we measure how likely app developers make mistakes when deploying App link verification. Second, we discuss a new vulnerability we discovered which allows malicious apps to skip user prompting when unverified App links are clicked. Malicious apps can exploit this to stealthily hijack arbitrary HTTP/HTTPS URLs in the mobile browser without user knowledge.

5.1 App Link Verification

We start by examining whether *link verification* truly protects apps from hijacking attacks. Since App link has not been introduced for *App2014*, all the http/https links in 2014 were unverified. In the following, we focus on apps in *App2016*. In total, there are 8,878 apps that register App links, involving 18,561 unique web domains. We crawled two snapshots of the association files for each domain in January and May of 2017 respectively. We use the January snapshot to discuss our key findings, and then use the May snapshot to check if the identified problems have been fixed.

Date	Apps w/ App Links	Apps Verif. Turned On	Apps Verified	Apps with Failed Verifications*					
				App Misconfig.	Host w/o Assoc. F.	Host w/ HTTP	Wrong Path	Host Invalid F.	Host Assoc. Other apps
Jan.17	8,878	415	194	26	177	11	0	10	60
May.17	8,878	415	192	26	171	8	0	18	57

Table 3: App Link verification statistics and common mistakes (App2016) based on data from January 2017 and May 2017. *One app can make multiple mistakes.

Type	Date	Hosts w/ Assoc. F.	Under HTTP	Wrong Path	Invalid File
iOS	Jan.17	12,570	1,817 (14%)	0 (0%)	108 (1%)
	May.17	13,541	1,820 (13%)	0 (0%)	113 (.8%)
Android	Jan.17	1,833	330 (18%)	4 (.2%)	81 (4%)
	May.17	2,779	474 (17%)	0 (0%)	118 (4%)

Table 4: Association files for iOS and Android obtained after scanning 1,012,844 domains.

Failed Verifications. As of January 2017, we find a surprisingly low ratio of verified App links. Among 8,878 apps that register App Links, only 194 apps successfully pass the verification (2%). More specifically, only 415 apps (4.7%) set the "*autoVerify*" field as TRUE, which triggers the verification process during app installation. This means the vast majority of apps (8,463, 95.3%) do not even start the verification process. Interestingly, 434 apps actually have the association file ready on their web servers, but the developers seem to forget to configure the apps to turn on the verification.

Even for apps that turn on the verification, only 194 out of 415 can successfully complete the process as of January 2017. Table 3 shows the common mistakes of the failed apps (one app can have multiple mistakes). More specifically, 26 apps incorrectly set the App link (*e.g.*, with a wildcard in the domain name), which is impossible for mobile OS to connect to. On the server-side, 177 apps turn on the verification, but the destination domain does not host the association file; 11 apps host the file under an HTTP server instead of the required HTTPS server; 10 apps' files are in invalid JSON format; 60 apps' association files do not contain the App link (or the app) to be verified. Note that for these failed apps, we do not distinguish whether they are malicious apps attempting to verify with a domain they do not own, or simply mistakes by legitimate developers.

We confirm all these mistakes lead to failed verifications by installing and testing related apps on a physical phone. We observe many of these mistakes are made by popular apps from big companies. For example, "`com.amazon.mp3`" is Amazon's official music app, which claims to be associated with "`amazon.com`". However, the association file under `amazon.com` does not certify this app. We tested the app on our phone, which indeed failed the verification.

In May 2017, we check all the apps again and find that most of the identified problems remain unfixed. Moreover, some apps introduce new mistakes: there are 8 more apps with an invalid association files in May compared to that of January. Manual examination shows that new mistakes are introduced when the developers update the association files.

Misconfigurations for iOS and Android. To show that App links verification can be easily misconfigured, we put together 1,012,844 web domains to scan their association files. These 1,012,844 domains is a union of Alexa top 1 million domains and the 18,561 domains extracted from our apps. We scan the association files for both Android and iOS.

As of January 2017, 12,570 domains (out 1 million) have iOS association files and only 1,833 domains have Android association files (Table 4). It is unlikely that there are 10x more iOS-exclusive apps. A more plausible explanation is iOS developers are more motivated to perform link verification, since iOS prohibits opening unverified HTTP/HTTPS links in apps. In contrary, Android leaves the decision to users by prompting users to choose between using apps or a browser.

We find iOS apps also have significant misconfigurations. This analysis only covers a subset of possible mistakes compared to Table 3, but still returns a large number. As of January 2017, 1817 domains (14%) are hosting the association file under HTTP, and there are additional 108 domains (1%) with invalid JSON files. One example is the Airbnb's iOS app. The app tries to associate with "`airbnb.com.gt`", which only hosts the association file under an HTTP server. This means users will not be able to open this link in the Airbnb app.

In May 2017, we scan these domains again. We observe 7.7% of increase of hosts with association files for iOS and 51.6% increase for Android. However, the number of misconfigured association files also increased.

5.2 Over-Permission Vulnerability

In addition to verification failures, we identify a new vulnerability in the setting preferences for App links. Recall

that unverified App links still have one last security defense — the end user. Android OS prompts users when unverified App links are clicked, and users can choose between a browser and the matched app. We describe an *over-permission vulnerability* that allows malicious apps to skip prompting for stealthy hijacking.

Over-Permission through Preference Setting. User prompting is there for better security, but prompting users too much can hurt usability. Android's solution is to take a middle ground using "preference" setting. When an App link is clicked, users can set "preference" for always opening the link in the native app without prompting again. We find that the preference setting gives excessive permissions. Specifically, the preference not only disables the prompting for the current link that the user sees, but all other (unverified) HTTP/HTTPS links that this app register. For example, if the user sets preference for "`https://bar.com`", all the links with "`https://`" in this app receive the permission. Exploiting this vulnerability allows malicious apps to hijack any HTTP/HTTPS URLs without alerting users.

Proof-of-Concept Attack. Suppose "`bar`" is a malicious app that register both "`https://bar.com`" and "`https://bank.com/transfer/*`". The user sets preference for using "`bar`" to open the link "`https://bar.com`", which is a normal action. Then without user knowledge, the permission also applies to "`https://bank.com/transfer/*`".

Later, suppose this user visits her bank's website in a mobile browser, and transfers money through an HTTPS request "`https://bank.com/transfer?sessionid=8154& amount=1000& recipient=tom`". Because of the preference setting, this request will automatically trigger `bar` without prompting the user. The browser wraps up this URL and the parameters in plaintext to create an Intent, and hands it over to the app `bar`. `bar` can then change the recipient and use the session ID to transfer money to the attacker. In this example, the attacker sets the path of the URI as "`/transfer/*`" so that `bar` would only be triggered during money transfer. The app can make this even stealthier by quickly terminating itself after the hijacking, and bouncing the user back to the bank website in the browser.

We validate this vulnerability in both Android 6.0.1 and 7.1.1 (the latest version). We implement the proof-of-concept attack by writing a malicious Android app to hijack the author's own blog website (instead of an actual bank). The attack is successful: the malicious app hijacked the plaintext parameters in the URL, and quickly bounced the user back to the original page in the browser. The bouncing is barely noticeable by users.

Discussion. Fundamentally, this vulnerability is caused by the excessive permission to unverified App links. When setting preferences, the permission is not applied to the *link-level*, but to the *scheme-level*. We suspect that the preference system of App links is directly inherent from scheme URLs. For scheme URLs, the preference is also set to the scheme level which makes more sense (*e.g.*, allowing the Facebook app to open all "`fb://`"). However, for App links, scheme-level permission means attackers can hijack any HTTP/HTTPS links.

To successfully exploit this vulnerability, a malicious app needs to trick users to set the preference (*e.g.*, using benign functionalities). For example, an attacker may design a recipe app that allows users to open recipe web links in the app for an easy display and sharing. This recipe app can ask users to set the preference for opening recipe links but secretly registers an online bank's App links to receive the same preference. We have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. We are currently working with the VRP team to mitigate the threat.

iOS has a similar preference setting, but not vulnerable to this over-permission attack. In iOS, if the user sets preference for one app to open an HTTPS link. The permission goes to all the HTTPS links that the app *has successfully verified*. The Android vulnerability is caused by the fact that permission goes to unverified links.

5.3 Summary of Vulnerable Apps

Thus far, our analysis shows that most apps are still vulnerable to link hijacking. First, scheme URLs are still heavily used among apps. Second, for apps that adopt App links, only 2% can pass the link verification. The over-permission vulnerability described above makes the situation even worse. In 2016, out of all 23,830 apps that adopt deep links, 23,636 apps either use scheme URLs or unverified App links. These are candidates of potential hijacking attacks.

6 Link Hijacking

While many apps are vulnerable in theory, the real question is how many vulnerable apps are exploited in practice? For a given app, how likely would other apps register the same URIs (*a.k.a.*, link collision)? Do link collisions always have a malicious intention? If not, how can we classify malicious hijacking from benign collisions?

To answer these questions, we first measure how likely it is for different apps to register the same URIs. Our analysis reveals the key categories of link collisions, and we develop a systematic procedure to label all of them. This analysis allows us to focus on the highly suspicious groups that are involved in malicious hijacking. Finally,

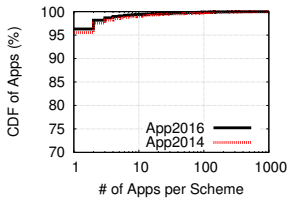


Figure 6: # of Collision apps per scheme.

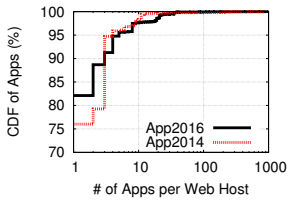


Figure 7: # of Collision apps per web host.

we present more in-depth case studies to understand the risk of typical attacks.

6.1 Characterizing Link Collision

Links collision happens when two or more apps register the same deep link URIs. When the link is clicked, it is possible for mobile OS to direct users to the wrong app. Note that simply matching "scheme" or app link "host" is not sufficient. For example, "`myapp://a/1`" and "`myapp://a/2`" do not conflict with each other since they use different "paths" in the URI. To this end, we define two apps have link collision only if there is at least one link that is opened by both apps.

Prevalence of Link Collisions. To identify link collision, we first group apps based on the scheme (scheme URL) or web host (App links). Figure 6 and Figure 7 show the number of apps that each scheme/host is associated with. About 95% of schemes are exclusively registered by one single app. The percentage is slightly lower for App links (76%–82%). Then for each group, we filter out apps that have no conflicting URIs with any other apps in the group, and produce apps with link collisions. Within *App2014*, we identify 394 schemes, 1,547 web hosts from 5,615 apps involved in link collisions. The corresponding numbers for 2016 are higher: 697 schemes and 3,272 web hosts from 8,961 apps.

Our result is a lower bound of actual collisions, biased towards popular apps. Schemes/hosts that are currently mapped to a single app might still have collisions with apps outside of our dataset. For the rest of our analysis, we focus on the more recent 2016 dataset.

Categorizing Link Collisions. We find that not all collisions have malicious intention. After manually analyzing these schemes and hosts, we categorize collisions into 3 types. Table 5 shows the top 10 mostly registered schemes/hosts and their labels.

- **Functional scheme (F)** is reserved for a common functionality, instead of a particular app. "`file`" is registered by 1,278 apps that can open files. "`geo`" is registered by 238 apps that can handle GPS coordinates. These schemes are expected to be registered by multiple apps. IANA [13] maintains a list of URI schemes, most of which are functional ones. This collision type does not apply to App links.
- **Per-app scheme/host (P)** is designated to an individual app. "`maps.google.com`" is to open Google Maps (but registered by 186 other apps) and "`fb`" is supposed to open Facebook app (but registered by 4 other apps). Collisions on per-app schemes/hosts are often malicious, with the exception if all apps are from the same developer.
- **Third-party scheme/host (T)** is used by third-party libraries, which often leads to (unintentional) link collision. "`x-oauthflow-twitter`" is a callback URL for Twitter OAuth. Twitter suggests developers defining their own callback URL, but many developers copy-paste this scheme from an online tutorial (unintentional collision). "`feedproxy.google.com`" is from a third-party RSS aggregator. Apps use this service to redirect user RSS requests to their apps (benign collision).

Scheme	Apps	Web Host	Apps
file (F)	1278	google.com (P)	480
content (F)	727	google.co.uk (P)	441
oauth (T)	520	zxing.appspot.com (T)	410
x-oauthflow-twitter (T)	369	maps.google.com (P)	187
x-oauthflow-espn-twitter (T)	359	beautygirlsinc.com (P)	148
zxing (T)	321	triposo.com (P)	131
testshop (T)	278	feeds.feedburner.com (T)	126
shopgate-10006 (T)	278	feeds2.feedburner.com (T)	123
geo (F)	238	feedproxy.google.com (T)	112
tapatalk-byo (T)	180	feedsproxy.google.com (T)	110

Table 5: Top 10 schemes and app link hosts with link collisions in App2016. We manually label them into three types: (F)= Functional, (P)= Per-App, (T)= Third-party

Because of the "shared" nature, functional schemes or third-party schemes/hosts are expected to be used by multiple apps. Related link collisions are benign or unintentional. In contrary, per-app schemes/hosts are (expected to be) designated to each app, and thus link collision can indicate malicious hijacking attempts.

6.2 Detecting Malicious Hijacking

Next, we detect malicious hijacking by labeling *per-app* schemes/hosts. This task is challenging since schemes and hosts are registered without much restriction—it is difficult to tell based on the name of the scheme/host. Our The high-level intuition is: 1) third-party schemes/hosts often have official documentations to teach developers how to use the library, which are searchable online; 2) functional schemes are

	Deep Links In Total	Link Collisions	After Pre-Processing	Functional	Third-party	Per-app
#Schemes (#Apps)	18,839 (20,257)	697 (7,432)	376 (6,350)	30 (2,135)	197 (3,972)	149 (893)
#Hosts (#Apps)	18,561 (8,878)	3,272 (2,868)	2,451 (2,083)	N/A	137 (999)	2,314 (1,593)

Table 6: Filtering and classification results for schemes and App link hosts (App2016).

well-documented in public URI standard. To these ends, we develop a filtering procedure to label per-app schemes/hosts. For any manual labeling tasks, we have two authors perform the task independently, and a third person to resolve any disagreements.

Pre-Processing. We start with the 697 schemes and 3,272 hosts (8,961 apps) that have link collisions in *App2016*. We exclude schemes/hosts where all the collision apps are from the same developer. This leaves 376 schemes and 2,451 web hosts for further labeling.

Classifying Schemes. We label schemes in two steps. The results are shown in Table 6. First, we filter out functional schemes. IANA [13] lists 256 common URI schemes, among which there are a few per-apps scheme under "provisional" status (*e.g.*, "`spotify`"). We manually filter them out and get 175 standard functional schemes. Matching this list with our dataset returns 30 functional schemes with link collisions. Then, to label third-party schemes, we manually search for their documentations or tutorials online. For certain third-party schemes, we also check the app code to be sure. In total, we identify 197 third-party schemes, and the rest 149 schemes are per-app schemes (also manually checked).

Figure 8 shows the number of collision apps for different schemes. Not surprisingly, per-app schemes have fewer collision apps than functional and third-party schemes.

Classifying App Link Hosts. This only requires labeling third-party hosts from per-app hosts. In total, there are 2,451 hosts after pre-processing. We observe that 1633 hosts are jointly registered by 5 apps, and 347 subdomains of "`google.com`" are registered by 2 apps. All these hosts are not third-party hosts, which helps to trim down to 471 hosts for manual labeling. We follow the same intuition to label third-party web hosts by manually searching their official documentations. In total, we label 137 third-party hosts, and 2,314 per-app hosts. Figure 9 compares per-app hosts and third-party hosts on their number of collision apps, which are very similar.

Testing Automated Classification. Clearly manually labeling cannot scale. Now that we have obtained the labels, we briefly explore the feasibility of automated classification. As a feasibility test, we classify per-app schemes from third-party schemes using 10 features such as unique developers per scheme, and apps per scheme (feature list in Appendix). 5-fold cross-validation using SVM and Random Forests classifiers return an accuracy of 59% (SVM) and 62% (RF). If we only focus on schemes that have a higher-level of collisions (*e.g.*, > 4 developers), it returns a higher accuracy: 84% (SVM) and 75% (RF). The accuracy is not high enough for practical usage. Intuitively, there are not many restrictions on how developers register their URIs, and thus it is possible that the patterns of per-app schemes are not that strong.

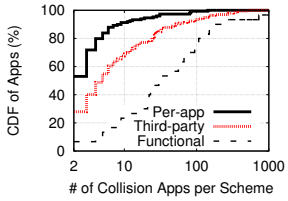


Figure 8: # of collision apps per scheme.

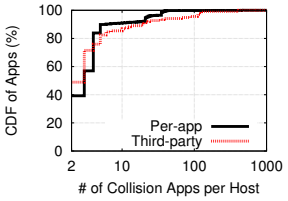


Figure 9: # of collision apps per host.

Since fully automated classification is not yet feasible, we then explore useful heuristics to help app market admins to conduct collision auditing. We rank features based on the information gain, and identify top 3 features: average number of apps from the same developer (apDev), number of unique no-prefix components (npcNum) and number of unique components (ucNum). Regarding apDev, the intuition is that developers are likely to use a different per-app scheme for each of their apps, but would share the same third-party schemes (*e.g.*, `oauth`) for all their apps. A larger apDev of the collision link indicates a higher chance of being a third-party scheme. Moreover, third-party schemes are likely to use the same component name for different apps (*i.e.*, less unique), leading to smaller npcNum and ucNum.

6.3 Hijacking Results and Case Studies

In total, we identify 149 per-app schemes and 2,314 per-app hosts that are involved in link collisions. The related apps (893 and 1,593 respectively) are either the attacker or victim in the hijacking attacks. To understand how per-app schemes and hosts are hijacked, we perform in-depth cases studies on a number of representative attacks.

Traffic Hijacking. We find apps that register popular websites' links (or popular apps' schemes) seeking to redirect user traffic to themselves. For example, "`google.com`" is registered by 480 apps from 305 non-Google developers. The scheme

"google.navigation" from Google Maps is hijacked by 79 apps from 32 developers. The intuition is that popular sites and apps already have a significant number of links distributed to the web. Hijacking their links are likely to increase the attacker apps' chance of being invoked. We find many popular apps are among the hijacking targets (*e.g.*, Facebook, Airbnb, YouTube, Tumblr). Traffic hijacking is the most common attack.

URL Redirector MITM. A number of hijackings are conducted by "URL Redirector" apps. When users click on an http/https link in the browser, these Redirector apps redirect users to the corresponding apps. Essentially, Redirector apps play the role of mobile OS in redirecting URLs, but their underlying mechanisms have several security implications. For example, URLLander (com.chestnutcorp.android.urlander) and AppRedirect (com.nevoxo.tapatalk.redirect) each has registered HTTPS links from 36 and 75 web domains respectively (unverified) and has over 10,000 installs. We suspect that users install Redirector apps because of the convenience, since these apps allow users to open the destination apps (without bouncing to the browser) even if the destination apps have not yet adopted App links. The redirection is hard coded without the consent of the destination apps or the originated websites.

URL redirector apps can act as man-in-the-middle (MITM) to hijack HTTP/HTTPS URLs. For example, URLLander registered "https://www.paypal.com" for redirection. When a user visits paypal.com using a browser (usually logged-in), the URL contains sensitive parameters including a SESSIONID. Once the user agrees to use URLLander for redirection, the URL and SESSIONID will be handed over to URLLander by the browser in plaintext. This MITM threat applies to all the popular websites that Redirector apps registered such as facebook.com, instagram.com, and ebay.com. Particularly for eBay, we find that the official eBay app explicitly does not register to open the link "payments.ebay.com", but this link was registered by Redirector apps. We analyze the code of AppRedirect and find it actually writes every single incoming URL and parameters in a log file. Redirection (and MITM) can be automated without prompting users by exploiting the over-permission vulnerability (see §5.2) — if the user once sets a preference for just one of those links.

Hijacking a Competitor's App. Many apps are competitors in the same business, and we find targeted hijacking cases between competing apps. For example, Careem (com.careem.acma) and QatarTaxi (com.qatar.qatartaxi) are two competing taxi booking apps in Dubai. Careem is more popular (5M+ downloads), which uses scheme "careem" for many functionalities such as booking a ride (from hotel websites) and adding credit card information. QatarTaxi (10K downloads) registers to receive all "careem://*" deep links. After code analysis, we find all these links redirect users to the QatarTaxi app's home screen, as an attempt to draw customers.

Dataset	App Link (Webpage)	Scheme URL (Webpage)	Intent URL (Webpage)
Alexa1M	3.2M (480K)	431K (197K)	1,203 (452)

Table 7: Number of deep links (and webpages that contain deep links) in Alexa top 1 million web domains.

Bad Scheme Names. Hijackings are also caused by developers using easy-to-conflict scheme names. For example, Citi Bank's official app uses "deeplink" as its per-app scheme, which conflicts with 6 other apps. These apps are not malicious, but may cause confusions — a user is going to open the Citi Bank app, but a non-related app shows up (and vice versa). We detect 14 poorly named per-app schemes (*e.g.*, "myapp", "app").

7 Mobile Deep Links on The Web

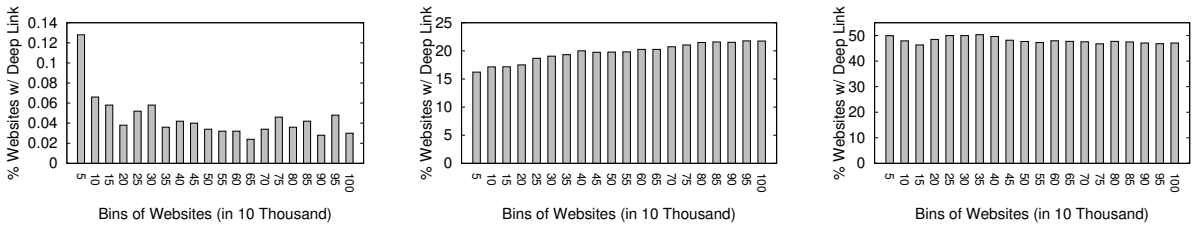
Our analysis shows that hijacking risks still widely exist within apps. Next, we move to the web-side to examine how mobile deep links are distributed on the web, and estimate the likelihood of users encountering hijacked links. In addition, we focus on *Intent URL* to examine its adoption and usage. We seek to estimate the impact of Intent URLs to mitigating hijacking threats.

In the following, we first measure the prevalence of Intent URLs on the web, and compare it with scheme URLs and App links. Then, we revisit the hijacked links detected in §6 and analyze their appearance on the web.

7.1 Intent URL Usage

Intent URL is a secure way of calling deep links from websites by specifying the target app's package name (unique identifier). In theory, Intent URL can be used to invoke existing app components defined by scheme URLs (and even App links) to prevent hijacking. The key question is how widely are Intent URLs adopted in practice.

Intent URLs vs. Other Links We start by extracting mobile deep links from web pages in *Alexa1M* collected in §3. For App links and scheme URLs, we match all the hyperlinks in the HTML pages with the link registration entries extracted from apps. We admit that this method is conservative as we only include deep links registered by apps in our dataset. But the matching is necessary since not all the HTTP/HTTPS links or schemes on the web can invoke apps. For Intent URLs, we identify them



(a) Intent URLs (b) Scheme URLs (c) App Links

Figure 10: Deep link distribution among Alexa top 1 million websites. Website domains are sorted and divided into 20 even-sized bins (50K sites per bin). We report the % of websites that contain deep links in each bin.

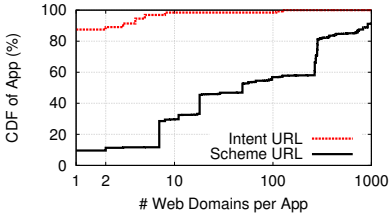


Figure 11: Number of websites that host deep links for each app.

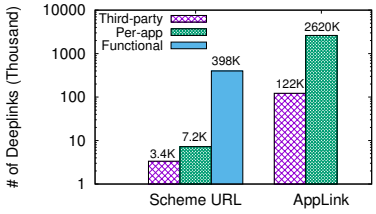


Figure 12: Different type of hijacked deep links in Alexa1M.

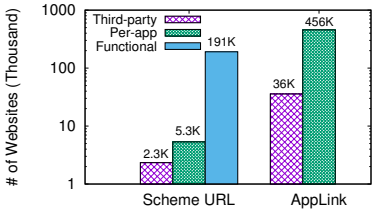


Figure 13: Webpages that contain hijacked deep links in Alexa1M.

based on their special format ("`intent://*;end`"). The matching results are shown in Table 7.

The key observation is Intent URLs are rarely used. Out of 1 million web domains, only 452 (0.05%) contain Intent URLs in their index page. As a comparison, App links and Scheme URLs appear in 480K (48%) and 197K (19.7%) of these sites. For the total number of links, Intent URL is also orders of magnitude lower than other links (1,203 versus 3.2M and 431K). This extremely low adoption rate indicates that Intent URLs have little impact to mitigating hijacking risks in practice.

Challenges to Intent URL Adoption. Since Android still supports scheme URLs, it is possible that developers are not motivated to use Intent URLs to replace the still-functional scheme URLs. In addition, even if security-aware developers use Intent URLs on their own websites, it is difficult for them to upgrade scheme URLs that have been distributed to other websites.

As shown in Figure 10(a), Intent URLs are highly skewed towards to high-ranked websites. In contrary, Scheme URLs are more likely to appear in low-ranked domains (Figure 10(b)), and App links' distribution is relatively even (Figure 10(c)). A possible explanation is that popular websites are more security-aware.

Then we focus on apps, and examine how many websites that contain an app's deep links (Figure 11). We find that most apps have their Intent URLs on a single website (90%). We randomly select 40+ pairs of the one-to-one mapped apps and websites for manual examination. We find that almost all websites (except 2) are owned by the app developers, which confirms our intuition. Scheme URLs are found in more than 5 websites for 90% of apps (50 websites for more than half of the apps). It is challenging to remove or upgrade scheme URLs across all these sites.

Insecure Usage of Intent URL. Among the 1,203 Intent URLs, we find 25 Intent URLs did not specify the package name of the target app (only the host or scheme). These 25 Intent URLs can be hijacked.

7.2 Measuring Hijacking Risk on Web

To estimate the level of hijacking risks on the web, we now revisit the hijacking attacks detected in §6 (those on per-app schemes/hosts). We seek to measure the volume of hijacked links among webpages, and estimation App link's contributions over existing risks introduced by scheme URLs.

Hijacked Mobile Deep Links. We extract links from *Alexa1M* that are registered by multiple apps, which returns 408,455 scheme URLs and 2,741,817 App links. Among them, 7,242 scheme URLs and 2,619,565 App links contain per-app schemes/hosts (*i.e.*, hijacked links).

The key observation is that App links introduce orders of magnitude more hijacked links than scheme URLs, as shown in Figure 12 (log scale y-axis). We further examine the number of *websites* that contain hijacked links. As shown in Figure 13, App links have a dominating contribution: 456K websites (out of 1 million, 45.6%) contains per-app App links that are subject to link hijacking. The corresponding number for scheme URL is 5.3K websites (0.5%).

App links, designed as the secure version of deep links, actually expose users to a higher level of risks. In-

tuitively, http/https links have been used on the web for decades. Once apps register App links, a large number of existing http/https links on the web are automatically interpreted as App links. This creates more opportunities for malicious apps to perform link hijacking.

Links Carrying Sensitive Data. To illustrate the practical consequences of link hijacking, we perform a quick analysis on the hijacked links with a focus on their parameters. A quick keyword search returns 74 sensitive parameter names related to authentications (*e.g.*, `authToken`, `sessionid`, `password`, `access_token`, full list in Appendix). We find that 1075 hijacked links contain at least one of the sensitive parameters. A successful hijacking will expose these parameters to the attacker app. This is just one example, and by no means exhaustive in terms of possibly sensitive data carried in hijacked links (*e.g.*, PII, location).

8 Discussion

Key Implications. Our results shed light on the practical challenges to mitigate vulnerable mobile deep links. First, scheme URL was designed for mixed purposes, including invoking a generic function (functional/third-party schemes) and launching a target app (per-app schemes). The multipurpose design makes it difficult to uniformly enforce security policies (*e.g.*, associating schemes to apps). A more practical solution should prohibit per-app schemes, while not crippling the widely deployed functional/third-party schemes on the web.

Second, App links and Intent URLs were designed with security in mind. However, their practical usage has deviated from the initial design. Particularly for App links, 98% of apps did not implement link verification correctly. In addition to various configuration errors, a more important reason is unverified links still work on Android, and developers are likely not motivated to verify links. As a result, App links not only fail to provide better security, but worsen the situation significantly by introducing more hijackable links.

Finally, the insecurity of deep links leads to a tough trade-off between security and usability. Mobile deep links were designed for usability, to enable seamless context-aware transitions from web to apps. However, due to the insecure design, mobile platforms have to constantly prompt users to confirm the links they clicked, which in turn hurts usability. The current solution for Android (and iOS) takes a middle ground, by letting users set "preference" for certain apps to disable prompting. We find this leads to new security vulnerabilities (over permission risk in §5.2) that allow malicious apps to hijack arbitrary HTTP/HTTPS URLs in the Android browser.

Legacy Issue. Android does not strongly enforce App link verification possibly due to the legacy issues. First, scheme URLs are still widely used on websites as discussed in §7. Disabling scheme links altogether would inevitably affect users' web browsing experience (*e.g.*, causing broken links [8]). Second, according to Google's report [11], over 60% of Android devices are still using Android 5.0 or earlier versions, which do not support App link verification. Android allows apps (6.0 or higher) to use verified App links while maintaining backward compatibility by not enforcing the verification.

Countermeasures. We discuss three countermeasures to mitigate link hijacking risks. In the short term, the most effective countermeasures would be disabling scheme URLs in mobile browsers and WebViews. Note that this is not to disable the app interfaces defined by schemes, but to encourage (force) websites to use Intent URLs to invoke per-app schemes safely. Android may also whitelist a set of well-defined functional schemes to avoid massively breaking functional links. For customized scheme URLs that are still used on the web, Android needs to handle their failure gracefully without severely degrading user experience. Second, prohibiting apps from opening unverified App links to prevent link hijacking. The drawback is that apps without a web front would face difficulties to use deep links — they will need to rely on third-party services such as Brach.io [4] or Firebase [5] to host their association files. Third, addressing the over-permission vulnerability (§5.2), by adopting more fine-grained preference setting (*e.g.*, at the host level or even the link level). This threat would also go away if Android strictly enforces App link verifications.

Vulnerability Notification & Mitigation. Our study identifies new vulnerabilities and attacks, and we are taking active steps to notifying the related parties for the risk mitigation.

First, regarding the over-permission vulnerability, we have filed a bug report through Google's Vulnerability Reward Program (VRP) in February 2017. As of June 2017, we have established a case and submitted the second round of materials including the proof-of-concept app and a demo of the attack. We are waiting for further responses from Google. Second, we have reported our findings to the Android anti-malware team and the Firebase team regarding the massive unverified App links and the misconfiguration issues. Details regarding their mitigation plan, however, were not disclosed to us. Third, as shown in §5.1, most of the misconfigured App links have not been fixed after 5 months. In the next step, we plan to contact the developers, particularly those of hijacked apps and help them to mitigate the configuration errors.

Limitations. Our study has a few limitations. First, our conclusions are limited to mobile deep links of Android. Although iOS takes a more strict approach to enforcing the link verification, it remains to be seen how well the security guarantees are achieved in practice. Our brief measurement in §5.1 already shows that iOS universal links also have misconfigurations. More extensive measurements are needed to fully understand the potential security risks of iOS deep links. Second, our measurement scope is still limited comparing to the size of Android app market and the whole web. We argue that data size is sufficient to draw our conclusions. By measuring the most popular apps (160,000+) and web domains (1,000,000), we collect strong evidence on the incompetence of the newly introduced linking mechanisms in providing better security. Third, we only focus on the link hijacking threat, because this is the security issue that App links and Intent URLs were designed to address. There are other threats related to web-to-mobile communications such as exploiting WebViews and browsers [20, 37], and cross-site request forgery on apps [27, 46, 50]. Our work is complementary to existing work to better understand and secure the web-and-app ecosystem.

9 Related Work

Inter-app Communication & Deep Links. Researchers have discovered various vulnerabilities in the inter-app communication mechanism in Android [19, 23] and iOS [52], which leads to potential hijacking and spoofing attacks. The fundamental issue is a lack of source and destination authentication [52]. In the context of app-to-app communication, attacks may cause permission escalation [15, 21] and sensitive data leakage [46]. Mobile deep links (*e.g.*, scheme URL) inherent some of these vulnerabilities when facilitating communications between websites and apps. Unlike web URLs whose uniqueness is guaranteed by the DNS, mobile deep links lack a similar, centralized entity for link registration and addressing. As a result, multiple apps may register the same link, leading to hijacking risks. Our work is complementary to existing work since we focus on large-scale empirical measurements, providing new understandings to how the risks are mitigated in practice.

Other recent works on mobile deep links focus on improving usability instead of security. Two systems are proposed to automatically generate deep links for apps via static and dynamic code analysis [38, 49].

Mobile Browser Security. In web-to-app communications, mobile browsers play an important role in bridging websites and apps, which can also be the target of attacks. For example, malicious websites may attack the browser using XSS [27, 50] and origin-crossing [52]. The threat also applies to customized in-app browsers (called WebView) [20, 37, 40, 51]. In our work, we focus hijacking threats to apps, a different threat model where browser is the not target.

Detection and Mitigation. Existing research has explored different approaches to detect vulnerabilities in app-to-app communications. On one hand, static code analysis leverages call graphs and flow analysis to detect information leakages [15, 26, 36, 45, 57] and vulnerable interfaces for inter-app communications [14, 32, 33, 34, 41, 42, 43]. On the other hand, dynamic analysis tracks information flow in the runtime which can capture attacks that would be otherwise missed by static analysis [24, 28, 30, 54, 56]. To remove and mitigate vulnerabilities, researchers propose to automatically generate app patches [39, 45, 58], enforce strict policies [16, 17, 31, 51, 59] and provide guidelines for writing safer apps [31]. Our work highlights the significant gap between a security solution and the practical impact in mitigating threats. Beyond technical solutions, other factors such as developer incentives and capabilities and mobile platform policies also play a big role.

10 Conclusion

In this paper, we conducted the first large-scale measurement study on mobile deep links across popular Android apps and websites. Our results showed strong evidence that the newly proposed deep link methods (App links and Intent URLs) fail to address the existing hijacking risks in practice. In addition, we identified new vulnerabilities and empirical misconfigurations in App links which ultimately expose users to a higher level of risks. Finally, we made a list of suggestions to countermeasure the link hijacking risks in Android. Moving forward, we plan to further investigate automated methods for hijacking detection, and conduct more extensive measurements on iOS deep links in the future.

Acknowledgments

The authors wish to thank the anonymous reviewers and our shepherd Manuel Egele for their helpful comments, and Bolun Wang for sharing the scripts to collect the meta data of Android apps. This project was supported by NSF grant CNS-1717028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

[1] Alexa. http://www.alexa.com.

[2] Android Intents with Chrome. https://developer.chrome.com/multidevice/android/intents.

[3] App programming guide for ios. https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html.

[4] Branch. https://developer.branch.io/.

[5] Firebase App Indexing. https://firebase.google.com/docs/app-indexing.

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

[7] Interacting with Other Apps. https://developer.android.com/training/basics/intents/filters.html.

[8] iOS 9.2 Update: The Fall of URI Schemes and the Rise of Universal Links. https://blog.branch.io/ios-9-2-redirection-update-uri-scheme-and-universal-links/.

[9] Support Universal Links. https://developer.apple.com/library/content/documentation/General/Conceptual/AppSearch/UniversalLinks.html.

[10] Smartphone apps crushing mobile web times. https://www.emarketer.com/Article/Smartphone-Apps-Crushing-Mobile-Web-Time/1014498, October 2016.

[11] Android platform versions. https://developer.android.com/about/dashboards/index.html, May 2017.

[12] AKHAWE, D., AND FELT, A. P. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Proc. of USENIX Security* (2013).

[13] AUTHORITY, I. A. N. Uniform resource identifier (URI) schemes. http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml, February 2017.

[14] BAGHERI, H., SADEGHI, A., GARCIA, J., AND MALEK, S. COVERT: Compositional analysis of Android inter-app permission leakage. *IEEE Transactions in Software Engineering* (2015).

[15] BOSU, A., LIU, F., YAO, D. D., AND WANG, G. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proc. of ASIACCS* (2017).

[16] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. XManDroid: A new Android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04* (2011).

[17] BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. of USENIX Security* (2013).

[18] CHEN, E. Y., PEI, Y., CHEN, S., TIAN, Y., KOTCHER, R., AND TAGUE, P. Oauth demystified for mobile application developers. In *Proc. of CCS* (2014).

[19] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in Android. In *Proc. of MobiSys* (2011).

[20] CHIN, E., AND WAGNER, D. Bifocals: Analyzing webview vulnerabilities in Android applications. In *Proc. of WISA* (2014).

[21] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege escalation attacks on Android. In *Proc. of ISC* (2011).

[22] EGELMAN, S., CRANOR, L. F., AND HONG, J. You've been warned: An empirical study of the effectiveness of web browser phishing warnings. In *Proc. of CHI* (2008).

[23] ELISH, K. O., YAO, D., AND RYDER, B. G. On the need of precise inter-app ICC classification for detecting Android malware collusions. In *Proc. of MoST* (2015).

[24] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS 32*, 2 (2014), 5.

[25] ENGLEHARDT, S., AND NARAYANAN, A. Online tracking: A 1-million-site measurement and analysis. In *Proc. of CCS* (2016).

[26] GORDON, M. I., KIM, D., PERKINS, J. H., GILHAM, L., NGUYEN, N., AND RINARD, M. C. Information flow analysis of Android applications in DroidSafe. In *Proc. of NDSS* (2015).

[27] HAY, R., AND AMIT, Y. Android browser cross-application scripting (cve-2011-2357). Tech. rep., July 2011.

[28] HAY, R., TRIPP, O., AND PISTOIA, M. Dynamic detection of inter-application communication vulnerabilities in Android. In *Proc. of ISSTA* (2015).

[29] INTERNATIONAL DATA CORPORATION (IDC). Smartphone OS Market Share. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`, November 2016.

[30] JING, Y., AHN, G.-J., DOUPÉ, A., AND YI, J. H. Checking intent-based communication in Android with intent space analysis. In *Proc. of ASIACCS* (2016).

[31] KANTOLA, D., CHIN, E., HE, W., AND WAGNER, D. Reducing attack surfaces for intra-application communication in Android. In *Proc. of SPSM* (2012).

[32] KLIEBER, W., FLYNN, L., BHOSALE, A., JIA, L., AND BAUER, L. Android taint flow analysis for app sets. In *Proc. of SOAP* (2014).

[33] LI, L., BARTEL, A., BISSYANDE, T. F. D. A., KLEIN, J., LE TRAON, Y., ARZT, S., RASTHOFER, S., BODDEN, E., OCTEAU, D., AND MCDANIEL, P. IccTA: detecting inter-component privacy leaks in Android apps. In *Proc. of ICSE* (2015).

[34] LIU, F., CAI, H., WANG, G., YAO, D. D., ELISH, K. O., AND RYDER, B. G. MR-Droid: A scalable and prioritized analysis of inter-app communication risks. In *Proc. of MoST* (2017).

[35] LIU, Y., SONG, H. H., BERMUDEZ, I., MISLOVE, A., BALDI, M., AND TONGAONKAR, A. Identifying personal information in internet traffic. In *Proc. of COSN* (2015).

[36] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. of CCS* (2012).

[37] LUO, T., HAO, H., DU, W., WANG, Y., AND YIN, H. Attacks on webview in the Android system. In *Proc. of ACSAC* (2011).

[38] MA, Y., LIU, X., DU, R., HU, Z., LIU, Y., YU, M., AND HUANG, G. DroidLink: Automated generation of deep links for Android apps. *CoRR abs/1605.06928* (2016).

[39] MULLINER, C., OBERHEIDE, J., ROBERTSON, W., AND KIRDA, E. PatchDroid: Scalable third-party security patches for Android devices. In *Proc. of ACSAC* (2013).

[40] MUTCHLER, P., DOUPÉ, A., MITCHELL, J., KRUEGEL, C., AND VIGNA, G. A large-scale study of mobile web app security. In *Proc. of IEEE MoST* (2015).

[41] OCTEAU, D., JHA, S., DERING, M., MCDANIEL, P., BARTEL, A., LI, L., KLEIN, J., AND LE TRAON, Y. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proc. of POPL* (2016).

[42] OCTEAU, D., MCDANIEL, P., JHA, S., BARTEL, A., BODDEN, E., KLEIN, J., AND LE TRAON, Y. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *Proc. of USENIX Security* (2013).

[43] RAVITCH, T., CRESWICK, E. R., TOMB, A., FOLTZER, A., ELLIOTT, T., AND CASBURN, L. Multi-App security analysis with FUSE: Statically detecting Android app collusion. In *Proc. of PPREW* (2014).

[44] ROWINSKI, D. Digital strategy: Why native apps versus mobile web is a false choice. `https://arc.applause.com/2016/09/13/native-apps-versus-mobile-web-decision/`, September 2016.

[45] SBÎRLEA, D., BURKE, M. G., GUARNIERI, S., PISTOIA, M., AND SARKAR, V. Automatic detection of inter-application permission leaks in Android applications. *IBM Journal of Research and Development 57*, 6 (2013), 10–1.

[46] SCHLEGEL, R., ZHANG, K., ZHOU, X., INTWALA, M., KAPADIA, A., AND WANG, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proc. of NDSS* (2011).

[47] STAROV, O., GILL, P., AND NIKIFORAKIS, N. Are you sure you want to contact us? quantifying the leakage of pii via website contact forms. In *Proc. of PETS* (2016).

[48] STATISTA: THE STATISTICS PROTAL. Number of available applications in the Google Play store from December 2009 to February 2016. http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2016.

[49] TANZIRUL AZIM, ORIANA RIVA, S. N. uLink: Enabling user-defined deep linking to app content. In *Proc. of Mobisys* (2016).

[50] TERADA, T. Attacking Android browsers via intent scheme urls. Tech. rep., March 2014.

[51] TUNCAY, G. S., DEMETRIOU, S., AND GUNTER, C. A. Draco: A system for uniform and fine-grained access control for web code on Android. In *Proc. of CCS* (2016).

[52] WANG, R., XING, L., WANG, X., AND CHEN, S. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proc. of CCS* (2013).

[53] WU, M., MILLER, R. C., AND GARFINKEL, S. L. Do security toolbars actually prevent phishing attacks? In *Proc. of CHI* (2006).

[54] XIA, M., GONG, L., LYU, Y., QI, Z., AND LIU, X. Effective real-time android application auditing. In *Proc. of IEEE S&P* (2015).

[55] XING, L., BAI, X., LI, T., WANG, X., CHEN, K., LIAO, X., HU, S.-M., AND HAN, X. Cracking app isolation on apple: Unauthorized cross-app resource access on MAC OS X and iOS. In *Proc. of CCS* (2015).

[56] YANG, K., ZHUGE, J., WANG, Y., ZHOU, L., AND DUAN, H. IntentFuzzer: Detecting capability leaks of Android applications. In *Proc. of ASIACCS* (2014).

[57] YANG, Z., YANG, M., ZHANG, Y., GU, G., NING, P., AND WANG, X. S. AppIntent: analyzing sensitive data transmission in Android for privacy leakage detection. In *Proc. of CCS* (2013).

[58] ZHANG, M., AND YIN, H. AppSealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications. In *Proc. of NDSS* (2014).

[59] ZHANG, Y., YANG, M., GU, G., AND CHEN, H. FineDroid: Enforcing permissions with system-wide application execution context. In *Proc. of SecureComm* (2015).

Appendix

Features for Classifying Schemes. Table 8 shows a list of features for classifying per-app schemes and third-party schemes in §6. These features are selected based on the intuition that third-party schemes are likely to be used by a larger variety of apps and developers, but are used for similar components in the third-party library

Sensitive Mobile Deep Link Parameters . Table 9 is a list of sensitive parameters identified in the mobile deep links from Alexa top 1 million websites. We exclusively focus on link parameters that are related to authentication. These parameter names are used in §7 to match hijacked deep links that carry sensitive data. We obtain this list by keyword searching and manual annotation. This is by no means an exhaustive list. The goal is provide examples to illustrate practical consequences of link hijacking attacks.

Feature	Description
aNum	Total # of apps
uDev	# of developers
cNum	Total # of components
ucNum	# of unique components
utcNum	# of unique third-party components
npcNum	# of unique components name (no prefix)
tDev	# of developers with third-party components
apDev	Average # of apps of the same developer
tDevP	% of third-party developers
ucP	% of unique components

Table 8: Features used for scheme classification.

access_token, actionToken, api_key, apikey, apiToken, Auth, auth_key, auth_token, authenticity_token, authkey, authToken, autologin, AWSAccessKeyId, cookie, csrf_token, csrfKey, csrfToken, ctoken, fk_session_id, FKSESSID, FOGSESSID, force_sid, formkey, gsessionid, guestaccesstoken, hkey, IKSESSID, imprToken, jsessionid, key, keycode, keys, LinkedinToken, live_configurator_token, LLSESSID, MessageKey, mrsessionid, navKey, newsid, oauth_callback, oauth_token, pasID, pass, pass_key, password, PHPSESSID, piggybackCookie, plkey, redir_token, reward_key, roken2, seasonid, secret_key, secret_perk_token, ses_key, sesid, SESS, sessid, sessid2b4f0b11dea2f7ae4bfff49b6307d50f, SESSION, session_id, session_rikey, sessionGUID, sessionid, sh_auth, sharedKey, SID, tok, token, uepSessionToken, vt_session_id, wmsAuthSign, ytsession

Table 9: Sensitive parameters in mobile deep links.