Adlib: Analyzer for Mobile Ad Platform Libraries

Sungho Lee
KAIST
South Korea
eshaj@kaist.ac.kr

Sukyoung Ryu
KAIST
South Korea
sryu.cs@kaist.ac.kr

ABSTRACT

Mobile advertising has become a popular advertising approach by taking advantage of various information from mobile devices and rich interaction with users. Mobile advertising platforms show advertisements of nearby restaurants to users using the geographic locations of their mobile devices, and also allow users to make reservations easily using their phone numbers. However, at the same time, they may open the doors for advertisements to steal device information or to perform malicious behaviors. When application developers integrate mobile advertising platform SDKs (AdSDKs) to their applications, they are informed of only the permissions required by the AdSDKs, and they may not be aware of the rich functionalities of the SDKs that are available to advertisements.

In this paper, we first report that various AdSDKs provide powerful functionalities to advertisements, which are seriously vulnerable to security threats. We present representative malicious behaviors by advertisements using APIs provided by AdSDKs. To mitigate the security vulnerability, we develop a static analyzer, Adlib, which analyzes Android Java libraries that use hybrid features to enable communication with JavaScript code and detects possible flows from the APIs that are accessible from third-party advertisements to device-specific features like geographic locations. Our evaluation shows that Adlib found genuine security vulnerabilities from real-world AdSDKs.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software defect analysis**.

KEYWORDS

Advertising Libraries, Malicious Advertisements, Advertisement Attacks, Android Hybrid Apps

ACM Reference Format:
Sungho Lee and Sukyoung Ryu. 2019. Adlib: Analyzer for Mobile Ad Platform Libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330562

1 INTRODUCTION

Rapid growth in the number of smartphone and tablet users has driven the popularity of mobile advertising. According to eMarketer, the global mobile ad spending was 16% of the total digital ad spending in 2013, it was 51.1% in 2016, and it is expected to be 70.1% in 2019 [18]. Mobile advertising provides unanticipated supports for both application (app) developers and users by taking advantage of various information from mobile devices and rich interaction with users. Developers can monetize their apps with mobile advertisements; because mobile devices are almost always-on and are next to users, mobile advertisements are more frequently revealed to users than other approaches. In addition, since mobile devices maintain diverse information such as geographic locations and contact addresses, and they provide powerful functionalities like SMS and phone calls, users can receive personalized services.

In order to support rich and powerful interaction with users for advertisements, *mobile advertising platform SDKs (AdSDKs)* provide hybrid mechanisms, which use JavaScript code for user interaction and native code for accessing device resources. Hybrid mechanisms also allow JavaScript code and native code to communicate with each other so that users can interact with JavaScript code to utilize device resources via native code. With AdSDKs written in native code, *mobile advertising platform libraries (AdLibraries)* written in JavaScript wrap the AdSDKs and provide common APIs to advertisements written in JavaScript cross multiple mobile platforms. In addition, because various mobile advertising platforms provide their own sets of APIs, Interactive Advertising Bureau (IAB) developed Mobile Rich Media Ad Interface Definitions (MRAID) [23], a set of APIs for mobile advertisements. It is a standardized set to help advertisement developers (publisher) use the same APIs for different mobile advertising platforms.

Unfortunately, the richer the APIs mobile advertising platforms provide, the bigger the security threats of mobile advertisements exist. While MRAID defines common APIs for multiple mobile advertising platforms, each platform defines additional APIs in its AdSDK for its own purposes. Because such APIs are available not only to AdLibraries but also to advertisements, they may lead to security threats like fracking attacks [20]. Exploiting leaked APIs, advertisements can steal device information or perform malicious behaviors. Note that exploiting AdSDKs can incur much more effects than malware, since a vulnerable AdSDK can be integrated by multiple apps.

In this paper, we report that various widely-used AdSDKs provide powerful functionalities to advertisements possibly unintentionally, which are seriously vulnerable to security threats. When installing and executing an app integrated with such an AdSDK, advertisements may conduct malicious behaviors using the leaked APIs. To show the severity of such leaked APIs, we present representative malicious behaviors by advertisements using the APIs. Then, to

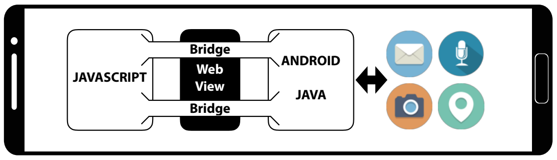


Figure 1: Overview of the Android hybrid communication

mitigate the security vulnerabilities, we develop a static analyzer, Adlib, which analyzes mobile AdSDKs that use hybrid features to enable communication with JavaScript code. While analysis techniques in this paper are language-agnostic, Adlib handles Android AdSDKs for now. In our evaluation, Adlib found genuine security vulnerabilities from real-world AdSDKs.

The contributions of this paper include the following:

- We report security vulnerabilities due to widely-used AdSDKs for the first time. The powerful APIs provided by AdSDKs may be available for arbitrary advertisements. We present concrete attack scenarios on sample apps provided by AdSDKs.
- We develop Adlib, which statically analyzes Android hybrid libraries and detects representative vulnerable patterns. We made them publicly available.
- We evaluated Adlib with real-world AdSDKs and found that AdSDKs indeed provide dangerous APIs that can incur severe security vulnerabilities.

2 BACKGROUND

2.1 Multi-language Communication in Hybrid Apps

Hybrid apps are one of the preferred approaches to support diverse mobile platforms. Native apps provide full device-specific features, but they require developers to build multiple apps in various native languages like Android Java, Swift, and C. On the contrary, web apps written in JavaScript are fully portable in the sense that they can run on any platforms that provide any browsers, but they cannot use device-specific features. Taking core aspects from each approach, hybrid apps support both JavaScript and device-specific native code, and their multi-language communication. By implementing main app logic in JavaScript, developers do not need to repeat the implementation for diverse platforms. Also, by allowing JavaScript code to access device resources via multi-language communication, users can enjoy powerful functionalities. We call multi-language communication *hybrid communication*.

Figure 1 illustrates an overview of the Android hybrid communication. Android Java is the native language of Android apps and `WebView` is a component that shows web pages and executes JavaScript code in Android apps. For presentation brevity, we use Android Java and Java interchangeably. In a hybrid app, `WebView` has one or more "bridges" and JavaScript code communicates with Java code to accesses device resources through the bridges. To implement Android hybrid apps, developers should attach bridges, which handle communication between Android Java and JavaScript, to `WebView`. Figure 2 shows an example: (a) is Android Java code, and (b) is JavaScript code in `http://www.example.com/index.html`.

```
1  class JSObj {
2    WebView wv;
3    @JavascriptInterface
4    public void sendMsg(String msg) { ...
5      wv.loadUrl("javascript:jsCb('send: ' + msg)");
6    }
7    public String getInfo() { return "example"; }
8  }
9    @Override
10   protected onCreate(Bundle savedInstanceState) {
11     WebView wv = new WebView(this);
12     wv.getSettings().setJavaScriptEnabled(true);
13     wv.addJavascriptInterface(new JSObj(wv), "brg");
14     wv.loadUrl("http://www.example.com/index.html");
15   }
```

(a) Android Java code

```
1  function jsCb(String msg){ ... }
2  brg.sendMsg("example");
3  var info = brg.getInfo();
```

(b) JavaScript code in `http://www.example.com/index.html`

Figure 2: Sample Android hybrid communication

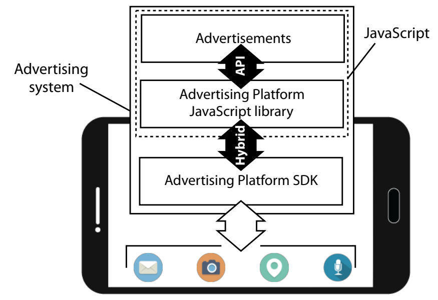
Because `WebView` does not execute JavaScript code by default, `setJavaScriptEnabled` (line 12) enables JavaScript to run on `WebView`. Then, `addJavascriptInterface` (line 13) attaches a bridge to `WebView`; it injects the Java object passed as its first argument, `new JSObj(wv)`, as the name passed as its second argument, `brg`, to the JavaScript environment that runs on `WebView`. We call such injected Java objects *injected objects*, and the named objects in the JavaScript environment *bridge objects*. Finally, on line 14, `loadUrl` loads the web page at a URL passed as its argument, and the JavaScript code in the web page runs on `WebView`.

JavaScript code can invoke Java methods through bridge objects like `brg`. Since Android 4.2, only public methods with the `@JavascriptInterface` annotation are accessible from JavaScript code. We call accessible Java methods from JavaScript *bridge methods*. When JavaScript code invokes a bridge method, it waits till the method finishes its execution. In addition to returning method call results, Java can pass values to the JavaScript environment by invoking `loadUrl` with an argument with the prefix "`javascript:`" as on line 5. In such cases, `loadUrl` recognizes the argument as JavaScript code and executes it asynchronously in the current JavaScript environment of `WebView`. Thus, the method call of `loadUrl` on line 5 executes the input JavaScript code, which calls the JavaScript function `jsCb` defined on line 1 in Figure 2(b).

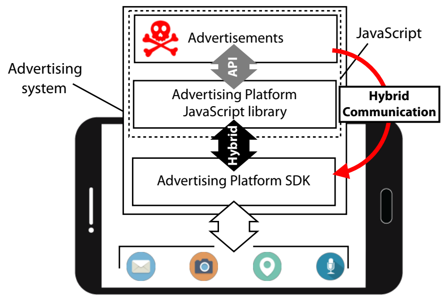
2.2 Mobile Advertising System

A mobile advertising system consists of three parts: 1) advertising platform SDKs, 2) advertising platform JavaScript libraries, and 3) advertisements. Figure 3(a) illustrates the overall architecture of a mobile advertising system. We call advertising platform SDKs and advertising platform JavaScript libraries *AdSDKs* and *AdLibraries*, respectively.

AdSDKs are implemented in native code such as Android Java for Android, and app developers may integrate AdSDKs in their apps to provide advertisements. Mobile advertising vendors often support their own AdSDKs with documents, which describe the ways to integrate AdSDKs into apps, to initiate AdSDKs, and to



(a) Mobile advertising system



(b) Advertisement threat model via AdSDK APIs

Figure 3: Mobile advertising system and a threat model

show advertisements in apps. When running an app, an AdSDK requests and receives an advertisement with an AdLibrary from its advertisement network and loads them; then, AdLibraries and advertisements are executed on `WebView`. AdLibraries communicate with AdSDKs using hybrid communication, and allow advertisements written in JavaScript to use device resources via APIs of AdLibraries.

Among various kinds of advertisements, two main kinds are a full-screen type and a banner type. Full-screen type advertisements cover the entire screen, and pause the running app until the advertisements finish their execution; they usually run for a short time. Banner type advertisements are generally displayed on the top or the bottom of the screen, and execute simultaneously with the running app; they often run for a long period. While advertisements are displayed on the screen, the JavaScript code of the advertisements run on `WebView` possibly utilizing device resources via AdLibraries.

3 MALICIOUS BEHAVIORS VIA ADSDK APIS

Most mobile advertising platforms contain simple but powerful AdSDKs written in native languages and complex but portable AdLibraries written in JavaScript. However, since AdSDK APIs are accessible not only from AdLibraries but also from advertisements, vulnerable AdSDKs can introduce serious security issues. In this section, we present four kinds of well-known malicious behaviors exploiting the APIs to show the severity of exposing AdSDK APIs to third-party advertisements.

3.1 Threat Model

The major threat comes from hybrid communication in the mobile advertising system. Figure 3(b) illustrates a threat model via bridges of AdSDKs. While advertisements should utilize device resources via AdLibraries, they can directly access powerful AdSDK APIs via hybrid communication.

In order for an advertisement to conduct malicious behaviors, an app integrated with an AdSDK that provides APIs accessing device resources must be already installed and run on an Android device. We assume that after the app runs, the AdSDK loads and displays the malicious advertisement. Because we can make a banner type advertisement, the app does not need to pause its execution while displaying the advertisement. In addition, the app developer must integrate the AdSDK into the app according to the instructions provided by the AdSDK vendor. Because AdSDK integration instructions contain their own requirements such as necessary permissions, the app must have all the necessary permissions that the AdSDK requires.

In the experiments, we load malicious advertisements on three AdSDKs: Millennial Media [36], Tapjoy [57], and ironSource [27]. For Millennial Media and Tapjoy, we use their sample apps provided by the vendors; because ironSource does not provide any sample app, we made an empty app and integrated it with the ironSource AdSDK according to the instructions in its documentation. We present malicious behaviors using our malicious advertisements without modifying any of the AdSDKs and their apps.

Since malicious advertisements can make real security issues, we use a proxy server, which intervenes between our device and a mobile advertising network server. When the device requests an advertisement, the proxy server intercepts the request and receives an advertisement from the advertising network server instead of the device. Then, the proxy server injects a payload that conducts malicious behaviors into the advertisement, and sends it to the device. Thus, the device receives a malicious advertisement containing the payload.

3.2 Representative Malicious Behaviors

Now, we show how AdSDK APIs can produce four kinds of outstanding malicious behaviors: *location tracking*, *DDoS attack*, *stealing app information*, and *file downloading*.

3.2.1 Location Tracking.

Location is one of the major private information because it allows apps to track mobile users and infer their activities. In order to protect location information from unauthorized apps, the Android system provides two permissions: `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. Unsurprisingly, most advertising platforms require one of the permissions to get location information to provide advertisements for items that are available nearby the users.

While such advertisements may be helpful for users, they may surpass their rights exploiting APIs available only for AdSDKs. Note that the permissions are granted for AdSDKs not for advertisements. The ability that advertisements can also access and utilize device resources via APIs of AdSDKs is unintentionally open to untrusted third-party advertisements.

An advertisement written in JavaScript can invoke a Java method that retrieves the location information via bridge objects that connect the Android Java and JavaScript as we discussed in Section 2.1. Indeed, Millennial Media provides several bridge objects to its AdLibrary, and they are accessible from advertisements. Consider the following code:

```
function printLoc(loc){
   console.log('lat: ' + loc.latitude + ', long: ' + loc.longitude);
}
var cbId = MmJsBridge.callbackManager.generateCallbackId(printLoc);
MmInjectedFunctionsMmjs.location('{"callbackId":' + cbId + '}');
```

One of the bridge objects provided by Millennial Media named `MmInjectedFunctionsMmjs` has a bridge method named `location`, which returns the current location of the device. The function `printLoc` is used as a callback function to receive the location information; in this example, it simply prints the latitude and longitude of the location on the console window, but it can send the location to an adversary's sever. The object `MmJsBridge.callbackManager` is defined in the AdLibrary of Millennial Media, which manages callback functions called in the Millennial Media AdSDK. The method `generateCallbackId` of the object gets a function as an argument and returns a corresponding callback identifier used to call the function in the AdSDK. After calling the function `location` with the callback identifier as a JSON format string, the AdSDK calls the `printLoc` callback with the location as a JSON object via `MmJsBridge.callbackManager`.

Advertisements can also track users' locations continuously. Because banner style advertisements can be displayed for a long period and can execute simultaneously with apps, such advertisements can get location information continuously using JavaScript built-in functions like `setInterval`, which invokes a given function repeatedly at interval of a given time.

While AdSDKs can send users' locations to publishers intentionally [54], they may send the information unintentionally. Because advertising networks are often targets of attackers who inject malicious code into legitimate advertisements [25, 65], the attackers can steal users' locations via AdSDK APIs.

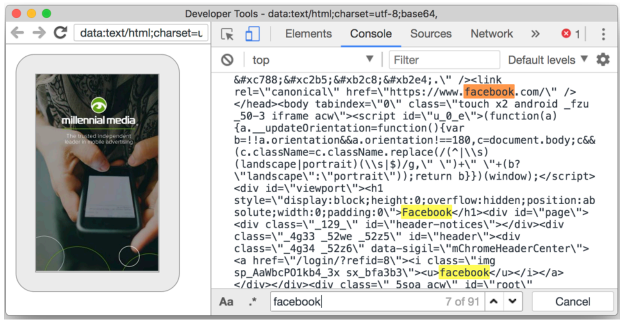
3.2.2 *DDoS Attack.*

Distributed Denial of Service Attack (DDoS attack), one of the most frequent cyber attacks, sends massive network traffic to a server from multiple devices to paralyze the services of the server. Now that many digital devices are connected to the Internet, the threat of the DDoS attack has been steadily increased. According to Ars Technica [58] and The Hacker News [45], the DDoS attack launched from smart devices is closed to 1 Tbps in 2016. Also, Cyberscoop [15] reported in its 2017 annual report that the DDoS attack is growing faster in size and complexity.

Mobile advertisement may be a sweet spot for the DDoS attack. While the threat of the DDoS attack becomes more severe as the number of botnets that send traffic to a server gets increased, because mobile advertisements are prevalent and they are displayed often and for a long period, adversaries can utilize a lot of mobile devices as botnets via mobile advertisements. Furthermore, frequently changed IP addresses of mobile devices can circumvent defense mechanisms banning specific IP addresses against the DDoS attack.

Consider the following code with Millennial Media again:

```
MmInjectedFunctionsMmjs.httpGet('{"url" : "www.example.com"}');
```



```
&#xc788;&#xc2b5;&#xb2c8;&#xb2e4;.\" /><link
rel=\"canonical\" href=\"https://www.facebook.com/\" />
</head><body tabindex=\"0\" class=\"touch x2 android _fzu
_50-3 iframe acw\"><script id=\"u_0_e\">(function(a)
{a.__updateOrientation=function(){var
b=!!a.orientation&&a.orientation!==180,c=document.body;c&&
(c.className=c.className.replace(/(^|\\s)
(landscape|portrait)(\\s|$)/g,\" \")+\" \"+(b?
\"landscape\":\"portrait\"));return b}})(window);</script>
<div id=\"viewport\"><h1
style=\"display:block;height:0;overflow:hidden;position:ab
solute;width:0;padding:0\">Facebook</h1><div id=\"page\">
<div class=\"_129_\" id=\"header-notices\"></div><div
class=\"_4g33 _52we _52z5\" id=\"header\"><div
class=\"_4g34 _52z6\" data-sigil=\"mChromeHeaderCenter\">
<a href=\"/login/?refid=8\"><i class=\"img
sp_AaWbcP01kb4_3x sx_bfa3b3\"><u>facebook</u></i></a>
</div></div><div class=\"_5soa acw\" id=\"root\"
```

Figure 4: Result of `httpGet` to `https://www.facebook.com`

The bridge object `MmInjectedFunctionsMmjs` has a bridge method named `httpGet`, which sends an `HTTP_GET` request to a server. When calling the method with a JSON format string including a target URL, the Millennial Media AdSDK sends an `HTTP_GET` request to the target URL using the mobile device resource. As with the location tracking attack, the AdSDK can send the request continuously using banner type advertisements.

In order to send the request, the app should have the `INTERNET` permission. Because every mobile advertising platform needs the `INTERNET` permission to receive advertisements, it is not an obstacle to perform the DDoS attack by mobile advertisements.

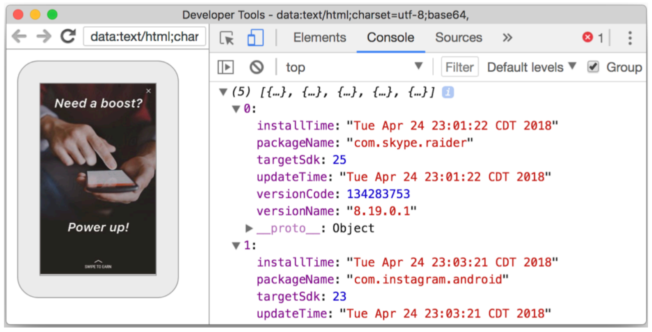
Note that it is not possible for advertisements to perform the DDoS attack using only JavaScript. There are two ways to use JavaScript to perform the DDoS attack: `XMLHttpRequest` and `iframe`. JavaScript code can invoke the `XMLHttpRequest` method repeatedly, and by setting the `src` attribute of an `iframe` to a server address multiple times, JavaScript code can send an HTTP request to the server repeatedly without interruption or notification. However, *Cross-Origin Read Blocking* of Chromium embedded in Android [13] forbids `XMLHttpRequests` from unauthorized origins, and the *X-Frame-Options* HTTP response header prevents servers from responding to requests from unauthorized origins. Even though Google, Facebook, and Netflix are protected by both defense mechanisms, advertisements are still able to perform the DDoS attack to servers via AdSDK APIs.

Figure 4 shows a screenshot of the Chrome debugger after sending an `httpGet` request and receiving its response. The left-hand side shows the device screen, and the right-hand side shows the Chrome debugger console window. The console window shows that the advertisement sends the HTTP request and receives its response successfully. As the device screen shows, the HTTP requests and responses using `httpGet` are not revealed to users without any interruption or notification.

3.2.3 *Stealing App Information.*

Even when new versions of apps are available, old versions of apps often still remain in mobile devices. Although most apps provide automatic updates to support more features or to fix security vulnerabilities for users' convenience, because software updates consume much network traffic and computation power, users may choose to update them manually or to defer the updates. Thus, apps may not be updated right after their new versions are released.

Unfortunately, such out-of-date apps may have 1-day vulnerabilities, which are good targets for adversaries [46]. When apps have



```
▼(5) [{…}, {…}, {…}, {…}, {…}]
  ▼0:
     installTime: "Tue Apr 24 23:01:22 CDT 2018"
     packageName: "com.skype.raider"
     targetSdk: 25
     updateTime: "Tue Apr 24 23:01:22 CDT 2018"
     versionCode: 134283753
     versionName: "8.19.0.1"
   ▶__proto__: Object
  ▼1:
     installTime: "Tue Apr 24 23:03:21 CDT 2018"
     packageName: "com.instagram.android"
     targetSdk: 23
     updateTime: "Tue Apr 24 23:03:21 CDT 2018"
```

Figure 5: Result of stealing app information

not yet applied patches to some security vulnerabilities, adversaries can take advantage of that fact and attack the apps using the vulnerabilities. Thus, detailed information about installed apps such as their version numbers may provide hints to adversaries.

However, AdSDKs often provide an API that reveals various information of installed apps on the device. Such information includes the package name, the installation time, the update time, and the version number of each installed app. While retrieving such information itself may not be malicious, it may leak the information to adversaries. For example, consider the following code:

```
AndroidWebViewJavascriptBridge._handleMessageFromAndroid
                              = function(info){console.log(info);};
AndroidJavascriptInterface.dispatchMethod(
                     '{"data": {"method":"getInstalledAppData"},
                       "callbackId":"none"}');
```

which uses the Tapjoy AdSDK to steal information of existing apps in mobile devices. The object `AndroidJavascriptInterface` is a bridge object. It has a bridge method named `dispatchMethod`, which sends a command to the Tapjoy AdSDK and returns the result of the command execution to a callback function. When calling the method with a JSON format string including the target method name `getInstalledAppData` and an arbitrary callbackId value, the Tapjoy AdSDK calls `_handleMessageFromAndroid` function of the `AndroidWebViewJavascriptBridge` object with the JSON format string including the information of installed apps received via the method `getInstalledAppData` of the AdSDK. As Figure 5 shows, advertisements can get all the information of the installed apps including their installation time, version names, package names, target AdSDKs, update time, and version code.

3.2.4 *File Downloading.*

Even an API defined in the standard MRAID [23] allows to download untrusted files:

> The `storePicture` method will place a picture in the device's photo album. The picture may be local or retrieved from the Internet. To ensure that the user is aware a picture is being added to the photo album, MRAID requires the SDK/container use an OS-level handler to display a modal dialog box asking that the user confirm or cancel the addition to the photo album for each image added. If the device does not have a native "add photo" confirmation handler, the SDK should treat the device as though it does not support `storePicture`.

While many mobile advertising platforms support this feature, some of them provide it in an unsafe way. Their AdSDKs allow advertisements to download any types of files not only images, and place downloaded files in different locations from the device's photo album. Also, they do not display a dialog box to ask users' confirmation. As Son *et al.* [54] reported, such behaviors enable advertisements to steal local files of the device they are running on through integrated AdSDKs. For example, consider the following code:

```
Android.saveFile('{"file" : "www.example.com/payload",
                   "path" : "malware"}');
```

which uses the ironSource AdSDK to download a malicious payload from a given URL. The ironSource AdSDK provides a bridge object named `Android`, which has a bridge method named `saveFile` that downloads a file from the Internet. When calling the function with a JSON format string including an online file address and a destination directory path, ironSource downloads the file and places it in the destination directory. Because this API does not show any confirmation window, users may not be aware of the downloaded file.

Although `saveFile` restricts locations where downloaded files are stored, malicious behaviors are still possible via the `saveFile` API. Among five storage options where apps can store their data [17], `saveFile` enforces downloaded files to be stored only in `external storage`, which is the app directory located in the external storage. However, because such downloaded files stored in the external storage are available to other apps, downloaded malicious files on a device can be used to steal local files of the device [54].

Furthermore, `saveFile` with other API functions together can change app behaviors. They can replace an existing file with a malicious file, which may make apps behave differently using the malicious file instead of the original file. For example, the following code:

```
Android.deleteFile('{"file" : "index.html",
                     "path" : "../assets"}');
```

can delete a file with a given name at a given location. After deleting it, advertisements can download a malicious file using `saveFile` to replace it with the malicious one.

4 ADLIB: ANALYZER FOR MOBILE ADSDKS

4.1 Overview

Adlib is an open-source static analyzer of AdSDKs, which detects vulnerable patterns that can be abused by advertisements. While malicious behaviors are performed by advertisements, Adlib analyzes AdSDKs rather than advertisements themselves due to two reasons. First, the main cause of the malicious behaviors does not come from individual advertisements but from APIs provided by AdSDKs. Even when an advertisement is safely verified by advertising vendors, *man-in-the-middle* attacks can make it malicious by injecting a payload into it via relaying between the advertising server and the device [16]. Second, collecting real-world mobile advertisements is challenging, because advertising platform vendors allow only verified developers or apps to receive advertisements. Furthermore, since most advertising platform vendors support only selected geographic regions, we can collect only a few advertisements from them.

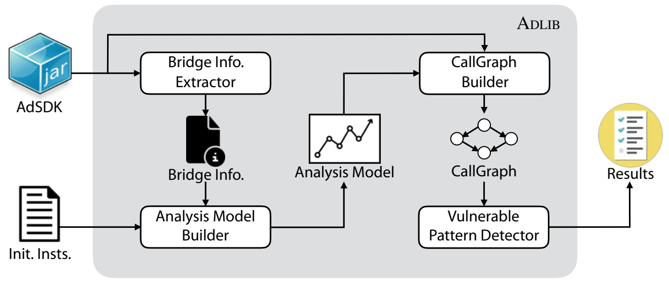


Figure 6: Overall architecture of Adlib

The overall architecture of Adlib is illustrated in Figure 6. The solid boxes denote modules and the arrows denote data flows between them. Adlib takes two inputs: an AdSDK archive and Init. Insts., a sequence of instructions that are necessary to execute to use the AdSDK. Most AdSDKs provide documents describing which instructions to execute to use them. Adlib first extracts bridge information from a given AdSDK via Bridge Info. Extractor. Then, Analysis Model Builder takes the bridge information and a given Init. Insts. and produces a flow-insensitive pointer analysis model of the AdSDK. Using the analysis model, CallGraph Builder constructs a call graph that represents control flows in the AdSDK. Finally, Vulnerable Pattern Detector analyzes data flows on the call graph and detects vulnerable patterns that may lead to malicious behaviors via advertisements.

4.2 AdSDK Analysis Model

Compared to analysis of Android apps, analysis of AdSDKs involves various technical challenges. First, because AdSDKs are libraries rather than standalone apps, they consist of library functions, analysis of which is different from conventional whole-program analysis. While a standalone app has a single entry point to start analysis, a library may have multiple entry points, one entry point for each library function. Furthermore, unlike whole-program analysis, which analyzes closed programs without any free identifiers, analysis of libraries must analyze library functions without any information from their callsites. Second, since an AdSDK supports multiple kinds of advertisements, it often creates multiple injected objects, one injected object for each kind of advertisement. We describe how Adlib addresses each technical challenge. Third, static analysis uses various context-sensitivities to reduce spurious paths, which is not effective to point out vulnerable APIs of AdSDKs. Since spurious paths blend an API's behavior with others, static analysis with low context-sensitivity may analyze that all APIs' behaviors are the same. However, high context-sensitivity not only reduces spurious paths, but also increases analysis overhead because it analyzes a method multiple times.

4.2.1 Virtual Entry Points for AdSDKs. To handle multiple entry points via library function calls in AdSDKs, Adlib additionally considers two kinds of methods: initialization methods and bridge methods. First, mobile apps display advertisements by calling initialization methods of AdSDKs, and advertising platform vendors provide AdSDKs with documents that describe how to set up and initialize the AdSDKs. Since such initialization methods are called

```
class Bridge{
  private Handler handler;
  public Bridge(Handler handler){
    this.handler = handler;
  }
  @JavascriptInterface
  public void open(String s){
    this.handler.open(s);
  }
}
void init(){
  if(isInterstitial)
    initForInterstitial(new HandlerForInterstitial());
  else
    initForBanner(new HandlerForBanner());
  Handler another = new HandlerForAnother(); ...
}
void initForInterstitial(Handler handler){
  Bridge bridge = new Bridge(handler);
  WebView wv = new WebView(this);
  wv.addJavascriptInterface(bridge, "bridge"); ...
}
void initForBanner(Handler handler){
  Bridge bridge = new Bridge(handler);
  WebView wv = new WebView(this);
  wv.addJavascriptInterface(bridge, "bridge"); ...
}
```

Figure 7: Example code of an AdSDK

by apps, Adlib considers initialization methods as entry points of AdSDKs. Second, bridge methods are also entry points in AdSDKs, because they are APIs accessible from the JavaScript side by AdLibraries and advertisements as described in Section 2.

Analysis Model Builder creates an entry point for both initialization methods and bridge methods. Initialization methods are given by Init. Insts. For example, the Init. Insts. of TapJoy is:

```
[ {"class":"Lcom/tapjoy/Tapjoy",
   "method":"connect(Landroid/content/Context;
                     Ljava/lang/String;
                     Ljava/util/Hashtable;
                     Lcom/tapjoy/TJConnectListener;)Z"},
  {"class":"Lcom/tapjoy/Tapjoy",
   "method":"onActivityStart(Landroid/app/Activity;)V"},
  {"class":"Lcom/tapjoy/TJPlacement",
   "method":"requestContent()V"},
  {"class":"Lcom/tapjoy/TJPlacement",
   "method":"showContent()V"} ]
```

which is a list of pairs of classes and methods used to initialize the AdSDK. The first three methods initialize bridge objects for advertisements, and the last method shows advertisements. Bridge methods declared in an AdSDK are extracted by Bridge Info. Extractor via the `@JavascriptInterface` annotation. Using them, Analysis Model Builder makes a fake method that calls all the methods with fake arguments of matching parameter types and CallGraph Builder uses the fake method as an entry to construct a call graph.

4.2.2 Object Abstraction for Multiple Ad Types. When an AdSDK creates multiple injected objects to support diverse kinds of advertisements, the behaviors of AdSDK APIs depend on their receiver objects, which are injected objects from the Android Java side. In other words, when a bridge method is called in the JavaScript side, the injected Java object corresponding to the bridge object determines the behavior of the bridge method. Figure 7 shows a sample code of

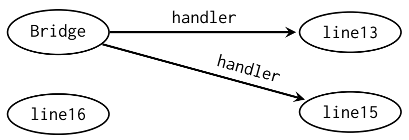


Figure 8: Object abstraction for multiple injected objects

an AdSDK. The `init` method declared on line 11 is the AdSDK initialization method; it calls `initForInterstitial` or `initForBanner` methods, passing an object of class `HandlerForInterstitial` and an object of class `HandlerForBanner`, respectively. When a full-screen type advertisement is loaded into the `WebView` created on line 20 and calls the `open` API of the `Bridge` class, the API calls the `open` method of the `HandlerForInterstitial` class, because the injected object is the one created on line 19. Similarly, when a banner type advertisement calls the same `open` API, it calls the `open` method of the `HandlerForBanner` class.

In order to analyze multiple injected objects in a sound manner, we use a more conservative abstraction for injected objects than for ordinary Java objects. While ADLIB uses creation-site abstraction for ordinary objects, it abstracts multiple injected objects of the same class as a single abstract object. Figure 8 shows our object abstraction model for the code in Figure 7. For simplicity, we show only injected objects and handler objects. Each ellipse denotes an object, and an arrow denotes that the source object has the target object as its field the name of which is the arrow label. While three handler objects are abstracted into individual objects based on their respective creation sites, lines 13, 15, and 16, two injected objects created on lines 19 and 24 are abstracted into a single `Bridge` abstract object. Because injected objects created on lines 19 and 24 have handler objects created on lines 13 and 15 in the `handler` field, respectively, the `Bridge` abstract object contains both handler objects as the value of its `handler` field. Thus, when ADLIB analyzes the `open` method on line 7 using the `Bridge` abstract object as its receiver, it analyzes two `open` methods of `HandlerForInterstitial` and `HandlerForBanner`.

4.2.3 Path Separation. Analysis of library functions often produces imprecise analysis results due to the lack of their calling contexts. Because such analyses consider any possible control flows conservatively, they lead to produce spurious control flows that do not exist in concrete program execution. Then, spurious flows among API functions usually make the behaviors of different API functions indistinguishable, losing analysis precision. In such cases, more precise analysis techniques like context sensitivity that distinguishes method call flows according to their call contexts may improve the analysis precision. However, at the same time, more precise analysis may incur more analysis performance overhead to maintain information of longer call flows, for example.

To distinguish control flows of APIs precisely without much performance overhead, we use a tag-based path separation mechanism. When ADLIB analyzes each API, it tags the method with a unique identifier. Then, when it analyzes method calls, it also tags callee methods with the identifiers of their callers. In other words, by propagating tags through method call flows, ADLIB distinguishes control flows of each API from those of non-API functions. Because this path separation mechanism is orthogonal to other sensitivity techniques, ADLIB can easily apply other techniques as well to eliminate spurious control flows further. Note that this mechanism also increases the performance overhead; because it analyzes each method reachable from n APIs n times, the overhead is proportional to the number of APIs. To reduce the overhead, we apply this mechanism to only app methods; we consider that all the Android and Java built-in methods have the same identifiers as their tags.

Table 1: Representative vulnerable patterns

Pattern	Representative case
a)	(File.<init>(x:File,y:String),y,x) $\hookrightarrow$ (File.delete(x:File),x)
b)	(URL.<init>(x:URL,y:String),y,x) $\hookrightarrow$ (URL.openConnection(x:URL),x,r) $\hookrightarrow$ (URLConnection.getInputStream(x:URLConnection),x,r) $\hookrightarrow$ (InputStream.read(x:InputStream,y:byte[]),x,y) $\hookrightarrow$ (FileOutputStream.write(x:FileOutputStream, y:byte[]),y)
c)	(PackageManager.getAppInfo(x:String,y:int,z:int),x)
d)	(HttpGet.<init>(x:HttpGet,y:String),y,x) $\hookrightarrow$ (HttpClient.execute(x:HttpHost,y:HttpGet),x)
e)	(Intent.<init>(x:Intent,y:String),y,x) $\hookrightarrow$ (Context.startActivity(x:Context,y:Intent),y)
f)	(LocationManager.getLastKnownLocation(x:String),ε,r) $\hookrightarrow$ (Location.getLatitude(x:Location),x)
g)	(Vibrator.vibrate(x:Vibrator,y:VibrationEffect),ε)

4.3 Detection of Vulnerable Patterns

We define the following seven vulnerable patterns:

a) *DeletingFile* deletes existing files in a mobile device;
b) *DownloadingFile* connects to the Internet, receives an input stream, reads data, and then writes data to a file;
c) *GettingAppInfo* retrieves various app information;
d) *RequestingHttp* sends an HTTP request to a server;
e) *LaunchingActivity* launches a new `Activity`;
f) *GettingLocation* retrieves location information of a mobile device or registers a listener that retrieves location information; and
g) *ControllingSensor* vibrates a device, or registers a listener that retrieves sensor information.

Since the patterns are closely related to security issues, they may lead to various kinds of malicious behaviors. For example, the *LaunchingActivity* pattern may lead to *Web to App Injection Attacks*, which abuse Android apps such as polluting app database, persistent storage, or app-specific logic by launching an `Activity` with malicious data from JavaScript code [22]. The pattern may also lead to *Task Hijacking Attacks*, which bring or inject a malicious `Activity` to a benign app's task [32, 48]. Note that four attacks described in Section 3 use five patterns: a), b), c), d), and f).

Table 1 shows representative vulnerable pattern cases. A vulnerable pattern case is a sequence of built-in API calls. A call $(m(x_1 : \tau_1, \cdots, x_n : \tau_n), \iota, o) \hookrightarrow (m'(x'_1 : \tau'_1, \cdots, x'_n : \tau'_n), \iota')$ denotes that there is a data flow from the entry of an AdSDK API to ι, which is either one of the arguments x_i of m or ε, and then from o to ι' where o is either x_i or the return value of m. For example, `(File.<init>(x:File,y:String),y,x)` $\hookrightarrow$ `(File.delete(x:File),x)`

Table 2: Transfer functions of the extended IFDS algorithm

Edge kind	Statement in method n	Transfer function
Ordinary	$x = y$	$\delta(d,\ \pi) = \begin{cases} \{d,\ x.f^*\} & \text{if } d = y.f^* \\ \{d\} & \text{otherwise} \end{cases}$
	$x.f = y$	$\delta(d,\ \pi) = \begin{cases} \{d,\ x.f.g^*\} \cup \boldsymbol{\{a.f.g^* \mid a \in aliases(\pi,\ x,\ n)\}} & \text{if } d = y.g^* \\ \{d\} & \text{otherwise} \end{cases}$
	$x = y.f$	$\delta(d,\ \pi) = \begin{cases} \{d,\ x.g^*\} & \text{if } d = y.f.g^* \\ \{d\} & \text{otherwise} \end{cases}$
Call-to-start	$x_r = m(x_1,\ \ldots,\ x_k)$	$\forall\ m_c(x'_1,\ \ldots,\ x'_k) \in targets(m).\ \delta_c(d,\ \pi) = \begin{cases} \{d,\ x'_i.f^*\} & \text{if } d = x_i.f^* \text{ for} 1 \le i \le k \\ \{d\} & \text{otherwise} \end{cases}$
Call-to-return-site	$x_r = m(x_1,\ \ldots,\ x_k)$	$\delta(d,\ \pi) = \{d\}$
Exit-to-return-site		$\forall$ return site $x_r = n(\ldots)\ \in\ m.$ $\delta_r(d,\ \pi) = \{d,\ x_r.f^* \mid \texttt{return}\ y;\ \in n\ \land\ d = y.f^*\}$ $\cup\ \boldsymbol{\{a.f^* \mid d = y.f^*\ \land\ a \in aliases(\pi,\ y,\ m)\}}$

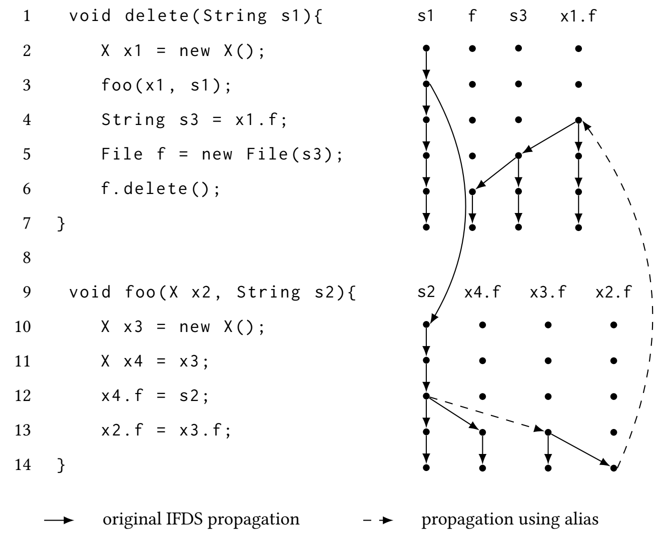

```
1    void delete(String s1){
2        X x1 = new X();
3        foo(x1, s1);
4        String s3 = x1.f;
5        File f = new File(s3);
6        f.delete();
7    }
8
9    void foo(X x2, String s2){
10       X x3 = new X();
11       X x4 = x3;
12       x4.f = s2;
13       x2.f = x3.f;
14   }
```

Figure 9: Extension of the IFDS algorithm to handle aliasing

denotes that there is a data flow from the entry of an AdSDK API to the second argument `y` of the constructor call of class `File`, and then from its first argument `x` to the sole argument of the call `File.delete(x:File)`. The sixth case shows that, as denoted by ε, `LocationManager.getLastKnownLocation` does not require any data flows from the entry of an AdSDK API; it just returns the location information regardless of its argument. The last case shows that just calling the `Vibrator.vibrate` method makes a device vibrate.

In order to detect the vulnerable patterns, the Vulnerable Pattern Detector module performs traditional data flow analysis on call graphs *from* arguments of vulnerable APIs *to* sensitive APIs that may lead to malicious behaviors using an extended IFDS algorithm [48]. While IFDS efficiently analyzes data flows using graph reachability, it does not handle data flows in alias relations. To handle aliases, we devise a simple extension of IFDS that provides precise analysis results for real-world AdSDKs as described below.

The IFDS algorithm takes three inputs: a *supergraph* consisting of control flow graphs for each method and call edges between them, a finite set of *data facts D*, and distributive *transfer functions* $\delta : S \times D \to 2^D$ where S is a statement. A data fact $d \in D$ is a variable followed by field accesses like $x.f^*$ in this paper. IFDS transfers data facts from an entry point to all program points by applying the transfer functions on the edges of the supergraph. A supergraph has four kinds of edges: *call-to-start* edges are from call statements to the entries of callees, *call-to-return-site* edges are from call statements to their return sites, *exit-to-return-site* edges are from exit nodes of callees to the return sites of callers, and *ordinary* edges are for the rest. IFDS uses different transfer functions for different kinds of edges to propagate data facts between statements. However, because it does not handle alias relations, it cannot collect data facts propagated via alias relations. Figure 9 shows a sample code with the initial data fact `s1`. Because IFDS performs a graph reachability analysis for each statement, a propagation edge connects `s1` to `s2` by the function call of `foo` on line 3. On line 12, a transfer function obtains two inputs—statement `x4.f = s2` and data fact `s2`—and returns a set of data facts, {`s2`, `x4.f`}, since the semantics of the assignment statement is to assign the value on the right to the left. Because it does not consider aliasing, it misses data fact `x3.f`, which comes from the alias between `x3` and `x4`.

We extend IFDS to consider alias relations by modifying the transfer functions. Table 2 shows the transfer functions of the extended IFDS algorithm for a simple Java-like language. The first and second columns describe edge kinds and statements respectively, and the third column describes the transfer functions δ with an input data fact d. For simplicity, we separate statements from inputs of the transfer functions. The original transfer functions take a statement and a data fact as inputs and produces a set of data facts to propagate to the next program point. To handle alias relations as well, we modify the transfer functions to take alias relations π as an additional input. The modified parts are in boldface font. Among ordinary edges, when the source node is an assignment statement to a field $x.f = y$, the transfer function propagates data facts to aliased local variables as well. It looks up local alias relations of x in the method n using π via *aliases*$(\pi,\ x,\ n)$, and joins them with the result of the original transfer function. Similarly, the transfer function of exit-to-return-site edges propagates data facts to aliased local variables in the caller method m of n via *aliases*$(\pi,\ x,\ m)$. Thus, each modification propagates data facts via intra- and inter-procedural alias relations, respectively. We use the call-to-return-site transfer function to propagate data facts on built-in API calls of vulnerable patterns. For example, on line 5 in Figure 9, `s3` is propagated to `f` by `(File.<init>(x:File,y:String),y,x)`. Thus, it finally matches the DeletingFile pattern on line 6 by `(File.delete(x:File),x)`.

4.4 Implementation

We built Adlib on top of HybriDroid [31], an open-source static analysis framework for Android hybrid apps. We also used WALA [24], another open-source static analysis framework for Java, Android Java, and JavaScript. We used HybriDroid as a third-party library to extract bridge information from a given Android Java bytecode, and WALA to build call graphs of Android AdSDKs. Utilizing various modules from two whole-program analyzers, Adlib constructs a library analysis model, which can be used for diverse client analyses.

We adopted the k-limiting approach to restrict the length of field access paths of data facts [29]. In the IFDS algorithm, the length of a field access path often increases infinitely due to recursion or loops, which causes the analysis not to terminate [33]. Thus, Adlib distinguishes only five field accesses in a field access path.

We used flow-insensitive pointer analysis results as alias relations taken by the extended IFDS. While flow-insensitive pointer analysis can produce imprecise results, the extended IFDS algorithm provides practically usable precision due to several reasons. First, *interprocedurally realizable paths* of IFDS prohibit propagation of data facts from return sites to spurious call sites. Second, because we use *static single assignment form (SSA)* [50] for all variables, variables have the same values regardless of program flows. Third, we filter out spurious data fact propagation using the types of fields in field access paths. Note that we can easily modify Adlib to use other alias analysis techniques such as backward tracking for more precise results.

Adlib can also analyze advertisements using AdSDKs and detect malicious behaviors in them. Since it is built on HybriDroid, it can analyze interaction between Android Java and JavaScript, thus it can analyze communication between an AdSDK and an advertisement. However, because collecting real-world advertisements is challenging, we analyzed the sample advertisements shown in Section 3 with Adlib, and it successfully analyzed them and correctly detected their accesses to the APIs that contain vulnerable patterns.

Note that since HybriDroid and WALA themselves are not sound, Adlib is not sound as well. We partially overcome this limitation by implementing additional modeling. In particular, we modeled the thread semantics for the `run` method of the `Runnable` object. For reflection, we modeled such cases when class names are given as string literals.

5 EVALUATION

In this section, we evaluate Adlib to see how well it detects vulnerable patterns in AdSDKs. For evaluation subjects, we collected 100 AdSDKs from 13 articles that discuss mobile advertising platforms from 2014 to 2017 [1–3, 9, 11, 21, 38–41, 55, 59, 60]. Among them, we filtered out 49 platforms that are designed to provide analytic information such as app usage patterns rather than advertisements. We also filtered out 16 platforms that we could not access, 6 platforms that do not have their own AdSDKs, and 5 platforms that do not use hybrid communication using `WebView` in their AdSDKs. Thus, we used 24 AdSDKs in our experiments [4–7, 12, 14, 19, 26, 27, 30, 34, 36, 37, 42–44, 49, 53, 56, 57, 61–64]. For each AdSDK, we downloaded the AdSDK jar file and extracted a sequence of instructions that are necessary to execute to use the AdSDK, from the AdSDK document, and constructed its Init. Insts. in a JSON format.

Table 3: Vulnerable patterns in AdSDKs

AdSDK (# of APIs)	Vulnerable Pattern (#)	TP	FP	CG (sec.)	Detect (sec.)
Appnext (31)	LaunchingActivity (1)	2	0	15.42	36.27
Cheetah Media Link (7)	LaunchingActivity (1)	1	0	1.29	1.02
CrossChannel (9)	LaunchingActivity (1)	1	0	39.83	2.46
InMobi (55)	DownloadingFile (2)	2	0	4450.83	613.11
	RequestingHttp (2)	2	0		
	LaunchingActivity (3)	3	0		
ironSource (60)	DeletingFile (2)	2	0	292.97	359.56
	DownloadingFile (1)	1	0		
	GettingAppInfo (1)	1	0		
	GettingLocation (1)	1	0		
	RequestingHttp (1)	1	0		
Leadbolt (6)	LaunchingActivity (1)	1	0	367.45	78.48
Millennial Media (51)	DownloadingFile (2)	2	0	374.15	3455.88
	GettingAppInfo (1)	1	0		
	GettingLocation (1)	1	0		
	ControllingSensor (1)	1	0		
	RequestingHttp (3)	3	0		
	LaunchingActivity (6)	6	0		
NativeX (22)	RequestingHttp (2)	1	1	3440.87	13854.07
	LaunchingActivity (16)	1	15		
Smaato (23)	LaunchingActivity (3)	3	0	257.24	179.80
StartApp (10)	GettingAppInfo (1)	1	0	289.22	35.27
	LaunchingActivity (2)	2	0		
Tapjoy (13)	GettingAppInfo (2)	2	0	571.72	91.59
	GettingLocation (2)	2	0		
	LaunchingActivity (2)	2	0		
	ControllingSensor (2)	2	0		
Upsight (60)	DeletingFile (2)	2	0	463.10	375.22
	DownloadingFile (1)	1	0		
	GettingAppInfo (1)	1	0		
	GettingLocation (1)	1	0		
	RequestingHttp (1)	1	0		
Verve (117)	DownloadingFile (2)	2	0	30.71	189.31
	GettingAppInfo (2)	2	0		
	RequestingHttp (2)	2	0		
Total (464)	**76**	**60**	**16**		

We analyzed the collected 24 AdSDKs with Vulnerable Pattern Detector and manually inspected the results. Table 3 summarizes the experimental results. The first column presents the AdSDKs and the numbers of their APIs that advertisements can access, the second column presents the vulnerable patterns with the numbers of APIs that may have the patterns, the third and the fourth columns present the numbers of true and false positives, and the last two columns present the analysis execution time of call graph construction and pattern detection, respectively; the time is the average of three runs.

Among 24 AdSDKs, Adlib reports that 13 AdSDKs contain 76 APIs that may have vulnerable patterns. Our manual inspection confirmed that 60 out of 76 APIs are true positives: 22 APIs contain the *LaunchingActivity* pattern, 10 for *RequestingHttp*, 8 for *DownloadingFile*, 8 for *GettingAppInfo*, 5 for *GettingLocation*, 4 for *DeletingFile*, and 3 for *ControllingSensor*. All 16 false positives come from a single AdSDK, NativeX. We manually confirmed that the false positives are due to the analysis imprecision of call graph construction for conditional branches. Because call graph construction using pointer analysis often handles opaque predicates imprecisely, data facts are propagated via spurious flows. We believe that applying pre- or post-analysis techniques for opaque predicates would reduce the number of false positives.

Now, we compare the techniques of Adlib with various existing techniques. Table 4 shows the comparison results of the path separation mechanism and the extension of the IFDS algorithm.

Table 4: Comparison of (a) the path separation with other context sensitivities and (b) the original IFDS and our extended IFDS

AdSDK	Insensitivity		1-CFA		2-CFA		3-CFA		1-OBJ		2-OBJ		3-OBJ		Pgm.	IFDS	Ext.	TP^+:
	Time	FP^+	Time	FP^+	Time	FP^+	Time	FP^+	Time	FP^+	Time	FP^+	Time	FP^+	Points	DFs	DFs	FP^+
Appnext	>12h	-	169.41	0	66.75	0	157.98	0	112.72	0	80.35	0	125.74	0	22823	12164	16443	-
Cheetah Media Link	2.33	0	2.29	0	2.67	0	3.06	0	2.95	0	4.17	0	5.20	0	2471	353	353	-
CrossChannel	11.18	0	7.41	0	9.59	0	13.00	0	7.99	0	8.02	0	12.73	0	25066	597	644	1 : 0
InMobi	>12h	-	>12h	-	1442.78	0	4910.83	0	>12h	-	>12h	-	>12h	-	201760	154156	178093	5 : 0
ironSource	522.80	3	365.91	1	659.17	1	870.71	0	>12h	-	>12h	-	>12h	-	55848	49053	73166	5 : 0
Leadbolt	578.09	1	260.71	0	434.96	0	585.07	0	674.13	1	768.89	1	3387.52	1	52345	33234	33487	1 : 0
MillennialMedia	8649.27	1	3725.15	1	1322.57	1	1429.20	1	6351.31	1	>12h	-	>12h	-	45684	37737	272885	8 : 0
NativeX	8671.77	14	8797.82	14	4314.43	12	10355.72	12	>12h	-	>12h	-	>12h	-	458594	317171	547256	2 : 16
Smaato	>12h	-	2141.02	0	1227.25	0	10164.49	0	448.10	0	1117.10	0	2937.22	0	45074	21675	31376	3 : 0
StartApp	164.72	4	191.13	1	81.52	0	238.96	0	71.08	0	142.75	0	162.60	0	57004	12822	13309	-
Tapjoy	745.01	1	479.27	0	290.66	0	829.31	0	559.24	0	1189.13	0	3055.87	0	93787	28441	32360	-
Upsight	671.94	34	444.86	1	361.28	1	733.08	0	>12h	-	>12h	-	>12h	-	74796	49643	73825	5 : 0
Verve	>12h	-	22164.93	27	>12h	-	1046.30	0	>12h	-	>12h	-	>12h	-	53393	47366	62170	6 : 0

The second to the 15th columns show the total analysis time in seconds and additionally reported false positives for widely-used context sensitive analyses. While Adlib with the path separation mechanism analyzes each AdSDK within three hours at most, some context sensitive analyses do not finish even in 12 hours, because of spurious data fact propagation caused by spurious paths in call graphs as well as the analysis overhead resulting from repeated analysis of a method for different contexts. Furthermore, since the context sensitivities do not completely distinguish behaviors of each API from others, all the analyses reported more false positives than the path separation mechanism.

The remaining four columns show the comparison results of the original IFDS algorithm and our extension. For each AdSDK, the 16th column shows the number of program points in it, the 17th and the 18th columns show the total numbers of data facts of the original IFDS and the extension, respectively, and the last column presents the true and false positives of the additionally reported vulnerable patterns by the extension. Except for MillennialMedia, the extension propagates at most two times more data facts while detecting more true vulnerable patterns than IFDS.

6 RELATED WORK

Luo *et al.* [35] reported possible attack scenarios on hybrid apps. They showed that `WebView` can serve as a channel for attacks and argued that `addJavascriptInterface` and callback methods of `WebView` break the sandbox of `WebView`. Bhavani [8] showed an attack that steals the cookie information of a web page or sends device information to an adversary's server silently. Jin *et al.* [28] reported possible code injection attacks on PhoneGap-based hybrid apps. Their approach puts data including malicious JavaScript code into PhoneGap apps via pre-defined APIs of PhoneGap. Hassanshahi *et al.* [22] reported attacks using a hyperlink, which can launch browsable `Activity` in the Android system. Their tool W2AIScanner could detect data flows from inputs of browsable `Activity` to sinks that may affect the app logic.

Georgiev *et al.* [20] reported attacks on PhoneGap-based apps via advertisement by using PhoneGap APIs that are accessible not only from app code but also from advertisements in an iframe. While their work is similar to ours, it focuses on only PhoneGap-based apps, and they showed only how many web pages access known APIs of PhoneGap. Son *et al.* [54] showed that some mobile advertising platforms allow advertisements to get location information, and advertisements can steal local files bypassing the Same Origin Policy exploiting a design flaw of AdSDKs. However, the design flaw is not in hybrid communication but in the `WebView` callback methods, and they experimented on only four AdSDKs.

To reduce attack surfaces from mobile advertising platforms, researchers have proposed to separate permissions of AdSDKs from those of apps [47, 52, 66]. This approach seems to be useful in preventing attacks by mobile advertising platforms, but they are inappropriate in defending against advertisement attacks because the permissions are required by advertising platforms instead of apps. Shehab and AlJarrah [51] designed a new permission system for Cordova-based hybrid apps. While their system grants permissions to each web page separately to restrict it to access only its necessary APIs, it is inappropriate for advertisement attacks as well.

Recently, researchers have started working on static analysis of Android hybrid apps. HybriDroid [31] is a static analysis framework that can detect representative bugs like `MethodNotFound` errors via analysis of inter-language communication. Brucker and Herzberg [10] also studied Android hybrid app analysis but they focused on Cordova-based apps.

7 CONCLUSION

Mobile advertising market is growing quickly, and many vendors provide their own platforms (AdSDKs) to support powerful advertisements for users. However, at the same time, untrusted third-party advertisements may exploit the functionalities provided by AdSDKs and make new threats for mobile apps. We show that advertisements can perform well-known malicious behaviors exploiting the powerful APIs provided by AdSDKs, and developed an open-source static analysis tool, Adlib, which analyzes real-world AdSDKs and detects APIs that are accessible from arbitrary advertisements. We evaluated Adlib with 24 widely-used AdSDKs, showed that they indeed provide 464 APIs accessible from advertisements, and confirmed that at least 60 APIs out of them have possibly vulnerable patterns. We believe that Adlib would be able to find not only vulnerable patterns in AdSDKs but also malicious behaviors in real-world advertisements, as our experiment with sample advertisements showed.

ACKNOWLEDGMENT

This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

[1] 60second marketer. 2015. Top 50 Mobile Marketing Tools and Platforms for Business. http://60secondmarketer.com/blog/2015/08/03/mobile-marketing-tools.

[2] AdPushup. 2015. Top 9 Best-Paying Mobile Ad Networks You Should Try. https://www.adpushup.com/blog/top-8-best-paying-mobile-ad-networks-you-should-try.

[3] AdSeeData. 2016. 2016 Index: AdMob Hit Mobile Advertising SDK Distribution. https://www.adseedata.com/insights/q3-2016-index-admob-hit-mobile-advertising-sdk-distribution_banner.

[4] Airpush. 2018. http://www.airpush.com.

[5] Appnext. 2018. https://www.appnext.com.

[6] Appodeal. 2018. https://www.appodeal.com.

[7] Appsfire. 2018. http://appsfire.com.

[8] AB Bhavani. 2013. Cross-site scripting attacks on android webview. *arXiv preprint arXiv:1304.7451* (2013).

[9] Blognife. 2017. Top 10 Mobile Interstitial Ad Networks List of 2017. https://blognife.com/2017/08/27/top-10-mobile-interstitial-ad-networks-list-2017.

[10] Achim D Brucker and Michael Herzberg. 2016. On the static analysis of hybrid mobile apps. In *International Symposium on Engineering Secure Software and Systems*. Springer, 72–88.

[11] Buzinga. 2014. 10 Most Popular Push Notification Services. http://www.buzinga.com.au/buzz/mobile-push-notification-services.

[12] Chartboost. 2018. https://www.chartboost.com.

[13] Chromium. 2018. Cross-Origin Read Blocking for Web Developers. https://www.chromium.org/Home/chromium-security/corb-for-developers.

[14] CrossChannel. 2018. http://www.crosschannel.com.

[15] Cyberscoop. 2017. Arbor: DDoS attacks growing faster in size, complexity. https://www.cyberscoop.com/ddos-attacks-growing-arbor-networks.

[16] Yvo Desmedt, Claude Goutier, and Samy Bengio. 1987. Special uses and abuses of the Fiat-Shamir passport protocol. In *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 21–39.

[17] Android Developer. 2018. Storage Options. https://developer.android.com/guide/topics/data/data-storage.html.

[18] eMarketer. 2015. Mobile Ad Spend to Top $100 Billion Worldwide in 2016, 51% of Digital Market. http://www.emarketer.com/Article/Mobile-Ad-Spend-Top-100-Billion-Worldwide-2016-51-of-Digital-Market/1012299.

[19] Flurry. 2018. https://developer.yahoo.com/flurry.

[20] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. 2014. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium*, Vol. 2014. NIH Public Access, 1.

[21] Guestpostblogging. 2017. Top 10 High Paying Mobile Ad Networks for Monetizing. http://guestpostblogging.com/top-10-high-paying-mobile-ad-networks-monetizing.

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

[23] IAD. 2018. Mobile Rich Media Ad Interface Definitions (MRAID). https://www.iab.com/guidelines/mobile-rich-media-ad-interface-definitions-mraid.

[24] IBM. 2006. T.J. Watson Libraries for Analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[25] InfoWorld. 2011. Cyber criminals strike at ad networks – again. https://www.infoworld.com/article/2623632/malware/cyber-criminals-strike-at-ad-networks----again.html.

[26] InMobi. 2018. http://www.inmobi.com.

[27] ironSource. 2018. http://www.ironsrc.com.

[28] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. 2014. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 66–77.

[29] Neil D Jones and Steven S Muchnick. 1979. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 244–256.

[30] Leadbolt. 2018. https://www.leadbolt.com.

[31] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 250–261.

[32] Sungho Lee, Sungjae Hwang, and Sukyoung Ryu. 2017. All about activity injection: threats, semantics, and detection. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 252–262.

[33] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In *Proceedings of the 30st IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 619–629.

[34] Cheetah Media Link. 2018. http://www.cheetahmedialink.com.

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

[36] MillennialMedia. 2018. http://www.millennialmedia.com.

[37] MobFox. 2018. http://www.mobfox.com.

[38] Mobuzz. 2016. Mobile Affiliate Networks. http://www.mobuzz.org/rankings/mobile-affiliate-networks.

[39] mobyaffiliates. 2014. Mobile and in-App Advertising Trends 2014. http://www.mobyaffiliates.com/blog/top-mobile-marketing-trends-2014.

[40] mobyaffiliates. 2016. Top Mobile Ad Networks 2016. http://www.mobyaffiliates.com/blog/top-mobile-ad-networks-2016.

[41] mobyaffiliates. 2016. Top Mobile Ad Servers 2016. http://www.mobyaffiliates.com/blog/top-mobile-ad-servers-2016.

[42] MoPub. 2018. https://www.mopub.com.

[43] NativeX. 2018. http://www.nativex.com.

[44] Facebook Audience Network. 2018. https://developers.facebook.com/products/app-monetization/audience-network.

[45] The Hacker News. 2016. World's largest 1 Tbps DDoS Attack launched from 152,000 hacked Smart Devices. http://thehackernews.com/2016/09/ddos-attack-iot.html.

[46] Jeongwook Oh. 2009. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. *Black Hat. Black Hat* (2009).

[47] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. 2012. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 71–72.

[48] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.

[49] Revmob. 2018. https://www.revmobmobileadnetwork.com.

[50] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 12–27.

[51] Mohamed Shehab and Abeer AlJarrah. 2014. Reducing attack surface on Cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*. ACM, 1–8.

[52] Shashi Shekhar, Michael Dietz, and Dan S Wallach. 2012. Adsplit: Separating smartphone advertising from applications. In *Proceedings of the the 21st USENIX Security Symposium*. 553–567.

[53] Smaato. 2018. https://www.smaato.com.

[54] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What mobile ads know about mobile users. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*.

[55] SOOMLA. 2016. Top 20 Mobile Ad Networks 2016. http://blog.soom.la/2016/01/top-20-mobile-ad-networks.html.

[56] StartApp. 2018. http://www.startapp.com.

[57] Tapjoy. 2018. http://home.tapjoy.com.

[58] Ars Technica. 2016. Record-breaking DDoS reportedly delivered by >145k hacked cameras. https://arstechnica.com/security/2016/09/botnet-of-145k-cameras-reportedly-deliver-internets-biggest-ddos-ever.

[59] Tune. 2015. Top 25 Advertising Partners, Spring 2015. https://www.tune.com/blog/top-25-advertising-partners-spring-2015-edition.

[60] Tune. 2016. Top 25 global advertising partners of 2016. https://www.tune.com/blog/top-25-global-advertising-partners-2016.

[61] UnityAds. 2018. https://unity3d.com/services/ads.

[62] Upsight. 2018. https://www.upsight.com.

[63] Verve. 2018. http://www.vervemobile.com.

[64] Vungle. 2018. https://vungle.com.

[65] ZDNet. 2018. Hackers target ad networks to inject cryptocurrency mining scripts. https://www.zdnet.com/article/hackers-now-mining-cryptocurrency-by-invading-ad-networks.

[66] Xiao Zhang, Amit Ahlawat, and Wenliang Du. 2013. AFrame: isolating advertisements from mobile applications in Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 9–18.