Understanding JavaScript Vulnerabilities in Large Real-World Android Applications

Wei Song, *Member, IEEE*, Qingqing Huang, and Jeff Huang, *Member, IEEE*

Abstract—JavaScript-related vulnerabilities are becoming a major security threat to hybrid mobile applications. In this article, we present a systematic study to understand how JavaScript is used in real-world Android apps and how it may lead to security vulnerabilities. We begin by conducting an empirical study on the top-100 most popular Android apps to investigate JavaScript usage and its related security vulnerabilities. Our study identifies four categories of JavaScript usage and finds that three of these categories, if inappropriately used, can respectively lead to three types of vulnerabilities. We also design and implement an automatic tool named JSDroid to detect JavaScript-related vulnerabilities. We have applied JSDroid to 1,000 large real-world Android apps and found that over 70 percent of these apps have potential JavaScript-related vulnerabilities and 20 percent of them can be successfully exploited. Moreover, based on the vulnerabilities identified by JSDroid, we have successfully launched real attacks on 30 real-world apps.

Index Terms—Android apps, JavaScript, WebView, security vulnerabilities, empirical study

1 Introduction

In recent years, hybrid mobile applications (apps) have become increasingly popular [1]. These apps are built with a mixture of mobile client languages, such as Java and C, and Web technologies, such as HTML, CSS, and JavaScript, thus allowing developers to reuse existing Web development solutions in different mobile platforms, e.g., Android, iOS, Windows, and so forth.

Along with the popularity of hybrid apps, the interaction between different languages broadens the attack surface of mobile apps. In particular, the dynamic nature of JavaScript makes it challenging to ensure that its interaction with the remainder of the system is safe and secure. Researchers have reported various JavaScript-related security vulnerabilities in popular apps. For example, the core *WebView* component (which enables JavaScript) may suffer from universal cross-site scripting (UXSS) attacks [2], malicious JavaScript code in a browser may perform remote code execution on user devices by invoking the Java code [3], and if JavaScript accesses to an execution context are inappropriately controlled, cross-zone JavaScript code in a local HTML file may access private data [4]. For an app with such vulnerabilities, a malware is not necessarily needed to launch the attack; sometimes a link (an HTML file that contains malicious JavaScript) is sufficient. If the user clicks the link in the app, then the malicious JavaScript is executed, and the app is attacked.

- *W. Song and Q. Huang are with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China. E-mail: wsong@njust.edu.cn, staryellow20151010@gmail.com.*
- *J. Huang is with the Department of Computer Science and Engineering, Texas A&M University, , College Station, TX 77843. E-mail: jeff@cse.tamu.edu.*

Manuscript received 28 Nov. 2017; revised 29 May 2018; accepted 5 June 2018. Date of publication 11 June 2018; date of current version 1 Sept. 2020. (Corresponding author: Wei Song.)
Digital Object Identifier no. 10.1109/TDSC.2018.2845851

Although JavaScript security in Android has attracted attention from the research community, little work has been performed to systematically study JavaScript usage and its vulnerabilities in real-world Android apps. For example, it is unknown how JavaScript is used in popular apps, what are the common usage patterns, what types of vulnerabilities can be exposed by using JavaScript, and how such JavaScript-related vulnerabilities can be exploited.

In this article, we present a systematic study of 1,000 large real-world apps from the Android market and report our findings. We begin by conducting a manual study of the top-100 most popular apps (e.g., *Gmail*, *Facebook*, and *Twitter*) and find that the uses of JavaScript can be categorized into only four types and that inappropriate uses of the first three types can respectively lead to three types of vulnerabilities. Based on our findings, we develop an automatic tool called JSDroid that can detect common JavaScript usage patterns in Android apps and reveal their potential vulnerabilities based on a static analysis. We apply JSDroid to 1,000 large real-world apps and find that 708 of these apps have potential JavaScript-related vulnerabilities and that 201 of them can be successfully exploited. Moreover, based on the identified vulnerabilities, we have successfully exploited 30 real-world apps.

Specifically, this paper provides two main contributions:

- We present a comprehensive characterization of JavaScript usage and JavaScript-related vulnerabilities in Android mobile apps based on an empirical study of the top-100 most popular Android apps. To our best knowledge, this work is the first comprehensive study of JavaScript security in large real-world Android hybrid apps. Our study not only sheds light on the current status of Android security but also provides insights to developers on how to

TABLE 1
APIs Related to WebView and its Customization

Class	Function	API
WebView	Web contents display	loadUrl() loadData() loadDataWithBaseURL()
	JavaScript-Java interaction	addJavascriptInterface()
	Get a settings manager	getSettings()
	Get an event handler	setWebViewClient()
WebSettings	JavaScript execution	setJavaScriptEnabled()
	File zone access	setAllowFileAccess() setAllowFileAccessFromFileURLs() setAllowUniversalAccessFromFileURLs()
WebViewClient	Page navigation	shouldOverrideUrlLoading()

appropriately use JavaScript in the development of Android apps.
- We present a static analysis tool named JSDroid that can automatically detect JavaScript-related vulnerabilities, and we apply it to 1,000 real-world Android apps. Our results show that newer Android versions reduce JavaScript-related vulnerabilities, but many existing devices are still vulnerable. JSDroid is open source and publicly available at http://bit.ly/myJSDroid.

The remainder of this article is organized as follows. Section 2 introduces preliminaries. Section 3 summarizes JavaScript usage patterns and related vulnerabilities, and Section 4 characterizes such patterns and vulnerabilities in the top-100 apps. Section 5 presents our evaluation on 1,000 large real-world apps. Section 6 proposes our exploitations of the identified JavaScript-related vulnerabilities in 30 real-world apps. Section 7 discusses how to reduce JavaScript vulnerabilities in practice. Section 8 reviews related work, and Section 9 concludes the paper.

2 Preliminaries

The use of JavaScript in Android apps is closely related to WebView, which is a browser-like view that displays Web contents and establishes the interaction between JavaScript and Java. In this section, we introduce WebView and its customization.

2.1 WebView

WebView is a subclass of the Android View class designed for displaying webpages and executing JavaScript. Prior to KitKat (Android 4.4), WebView used the WebKit[1] rendering engine to render webpages. Since KitKat, WebView has used Blink.[2]

Table 1 summarizes WebView-related classes and their APIs. There are two ways to create a WebView: one way is to include a `<WebView>` element in the layout XML file of an activity, and the other is to create a WebView object directly in the activity initialization. Once a WebView instance is created, a basic browser is built inside. The first three methods of WebView in Table 1 are used to display Web contents.

1. https://webkit.org/
2. https://www.chromium.org/blink

2.2 WebView Customization

Android provides two classes, *WebSettings* and *WebViewClient*, for developers to customize WebView: WebSettings manages the settings for WebView, e.g., whether JavaScript execution is enabled. WebViewClient is the event handler of WebView, which can specify the page navigation behavior of WebView.

2.2.1 JavaScript Execution and Zone Access

WebSettings is obtained by calling the `getSettings()` method of WebView. Developers can enable JavaScript execution and zone access using the following methods of WebSettings:

- `setJavaScriptEnabled(boolean flag)` enables or disables JavaScript execution in a webpage.
- `setAllowFileAccess(boolean flag)` enables or disables file access within WebView. This only determines the access to other parts of the file system, while the assets and resources of apps are always accessible via `file:///android_res` or `file:///android_asset`.
- `setAllowFileAccessFromFileURLs(boolean flag)` determines whether the JavaScript code running in the context of a file scheme URL can access contents from other file scheme URLs.
- `setAllowUniversalAccessFromFileURLs (boolean flag)` determines whether JavaScript running in the context of a file scheme URL can access contents from any other scheme (file, http, https) URLs.

Zones denote the scopes of JavaScript code, including file zones and http(s) zones. The last three methods of WebSettings can change the zone access ability of JavaScript. For example, scripts in a zone from one origin can be allowed to access files in zones from other origins. However, this may result in violations of the same origin policy (SOP) [5]. The origin of Web content is defined by its protocol (file, http, or https), domain, and port number. Two pages share the same origin if the above three elements are the same. If JavaScript in one origin reads or writes document object model (DOM) attributes in other origins, then SOP is violated.

2.2.2 Page Navigation

WebView can attach a WebViewClient via its method `setWebViewClient()`. The page navigation ability of

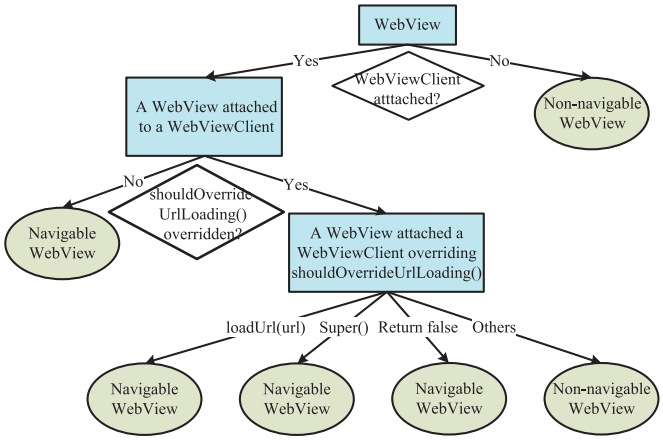


Fig. 1. Decision tree of WebView navigability.

WebView is determined by the method `shouldOverrideUrlLoading(WebView view, String url)` of WebViewClient which decides what action to take when users click a link in webpages loaded by WebView.

When users click a link, the default behavior is to launch the device browser. Developers can set WebView as navigable to load the destination URL in WebView. Fig. 1 illustrates the decision tree of this behavior. If the WebView does not attach a WebViewClient, then it is non-navigable; otherwise, the navigability depends on whether `shouldOverrideUrlLoading()` of WebViewClient is overridden. If the method is not overridden, is overridden but returns `super()` or a `false` value, or if it uses WebView to load the new URL via `loadUrl(url)`, then the WebView is navigable; otherwise, it is not navigable.

2.2.3 JavaScript-Java Interaction

Android provides a mechanism for building a bridge between JavaScript code in webpages and Java code in apps, allowing JavaScript code to invoke Java code. Specifically, the JS-to-Java bridge can be built by calling the method `addJavascriptInterface(Object object, String name)` of WebView which injects a Java object (the first parameter) into the JavaScript environment such that JavaScript code loaded by WebView can invoke methods of the Java object via the provided name (the second parameter).

3 Usage Patterns and Vulnerabilities

To understand JavaScript vulnerabilities, we conduct an empirical study of the top-100 most popular Android apps listed on Wikipedia.[3] Since the reference list shows more than 100 apps, we selected the first 100 available apps, including 92 free apps and 8 paid apps. We downloaded the Android Package (APK) files of the most recent versions of these apps from APK4Fun[4] and the Google Play Store[5] as of May 2016.

Table 2 lists these 100 apps including their names and versions. It took the authors approximately three person-months to manually analyze these apps. Since the source code of these apps is unavailable, we decompile their APK files into Java source code with the help of two tools, namely, Dex2Jar[6] and Apktool.[7] Android developers often obfuscate their apps by replacing the names of classes, methods, and variables with random symbols. This practice also makes the decompiled code difficult to read. Nevertheless, we could inspect the JavaScript code in these apps based on class names that are not obfuscated and the JavaScript APIs in Table 1.

Altogether, we identify four categories of JavaScript usage, namely, *local pattern*, *remote pattern*, *JSBridge pattern*, and *callback pattern*, as illustrated in Fig. 2. These four patterns cover almost all JavaScript-related APIs in existing Android SDKs. Moreover, we find that inappropriate uses of the first three patterns can lead to three types of JavaScript-related vulnerabilities, i.e., *file-based cross-zone vulnerability*, *WebView UXSS vulnerability*, and *JS-to-Java interface vulnerability*, respectively. These three types of vulnerabilities cover and classify all known vulnerabilities reported by existing research [2], [3], [4]. To our best knowledge, no approach classifies the vulnerabilities in this way. There are two threat models for attackers to exploit these vulnerabilities [3], [6]:

1) *Malicious third-party content:* The malicious content can be embedded in local HTML files from third-parties, local malware, third-party libraries, subframes (e.g., iframe), and remote websites, where it may call the app interfaces in some ways that the developers might not have forecasted.
2) *Network attack:* If an insecure network (e.g., insecure WiFi hotspot) is used, when an app requests a page via HTTP in WebView, a man-in-the-middle attacker can return any page that contains malicious JavaScript as the response.

From the developer's perspective, the first three JavaScript usage patterns share a common limitation: the JavaScript code loaded by WebView runs in a private background thread rather than the UI thread (main thread). To update UI elements, JavaScript code must be explicitly called in the UI thread. To address this limitation, starting with KitKat (Android 4.4), Google includes a WebView API (i.e., `evaluateJavascript(String script, ValueCallback <String> resultCallback))` that allows executing JavaScript in the UI thread. We refer to this type of JavaScript usage as the callback pattern (cf. Fig. 2d). Compared with the other three patterns, the callback pattern is more secure because the returned value from the JavaScript execution can only be passed to a callback function registered by the trusted Java code. We do not find any vulnerabilities related to the callback pattern.

Table 3 summarizes these patterns and their potential vulnerabilities. For each type of vulnerability, we also analyze the root cause and discuss the attack model for exploiting the vulnerability, as illustrated in Fig. 3. We present these results in detail in the following.

3.1 Local Pattern and File-Based Cross-Zone Vulnerability

3.1.1 Local Pattern

Developers often use WebView to load HTML files in app resources and dynamic JavaScript in app code. Since the

3. App List. http://en.wikipedia.org/wiki/List_of_most_downloaded_Android _applications (accessed in May 2016)
4. APK4Fun. http://www.apk4fun.com
5. Google Play Store. http://play.google.com
6. Dex2Jar. http://github.com/pxb1988/dex2jar
7. Apktool. http://code.google.com/p/android-apktool

TABLE 2
Top-100 Real-World Most Popular Android Apps

App name	Version	App name	Version	App name	Version	App name	Version
Gmail	5.0.1-1642443	*GoogleMaps*	8.4.1	*Facebook*	77.0.0.20.66	*YouTube*	11.07.59
Twitter	5.111.0	*GoogleTTS*	3.8.16	*GoogleSearch*	5.10.32.16	*WhatsAppMessenger*	2.16.95
Viber	6.0.1.13	*CleanMaster*	5.12.3	*ChromeBrowser*	51.0.2704.81	*GooglePlayGames*	3.7.23
Line	6.3.1	*AngryBirds*	5.2.0	*GoogleDrive*	2.4.181.13.34	*AvastSecurity*	5.2.0
Skype	7.01.0.669	*FruitNinjaFree*	2.3.4	*GoogleHangouts*	8.1.121732641	*GoogleStreetView*	1.8.1.2
TempleRun	1.6.1	*AdobeReader*	16.1.1	*FacebookMessenger*	72.0.0.16.67	*GoogleVoiceSearch*	2.1.4
Instagram	8.2.0	*GooglePlayBooks*	3.8.37	*GooglePlayMovies*	3.14.4	*TempleRun2*	1.24.0.1
ChatON	3.5.839	*MyTalkingTom*	3.4.1	*GooglePlayMusic*	6.9.2919B	*TalkingTomCat*	25.0.1
Shazam	6.5.1-160520	*SamsungPush*	1.6.00	*SubwaySurfers*	1.56.0	*PandoraRadio*	7.2.1
Tango	3.22.203462	*CandyCrushSaga*	1.76.1.1	*GoogleTranslate*	5.0.0.RC	*FreeAntiVirus*	5.3.0.2
Pou	1.4.69	*CandyCrushSoda*	1.67.7	*FacebookLite*	6.0.0.7.138	*GOLauncherEX*	2.10
Zedge	5.2.4	*Flipboard*	3.3.11	*AngryBirdsRio*	2.6.1	*PicsArtPhoto*	5.26.4
Dropbox	8.2.4	*HillClimbRacing*	1.29.0	*SuperBrightLED*	1.0.9	*DespicableMe*	2.7.3b
WeChat	6.3.18	*MXplayer*	1.7.40	*GoogleEarth*	8.0.2.2334	*FarmHeroesSaga*	2.52.5
OperaMini	16.0.2168	*KakaoTalk*	5.6.8	*HPPrintPlugin*	2.4-1.3.1-10e	*BarcodeScanner*	4.7.5
AdobeAIR	20.0.0.233	*PoolBilliardsPro*	3.3	*GoogleTalkback*	3.5.2	*GooglePlayNews*	3.5.2
eBay	2.8.2.1	*CutTheRope*	2.6.5	*AmazonKindle*	4.24.0.27	*SamsungPrintService*	2.19.16
BBM	2.13.1.14	*DUBatterySaver*	4.1.6	*ShootBubble*	3.42.52.5	*AngryBirdsGo!*	1.13.9
HayDay	1.28.146	*JetpackJoyride*	1.9.6	*ClashofClans*	8.332.9	*AngryBirdsSeasons*	6.1.1
Snapchat	9.30.5.0	*ESFileExplorer*	4.0.5	*GooglePhotos*	1.21.0.123	*Yahoo!Mail*	4.7.4
Kik	10.6.0.6560	*BatteryDoctor*	5.11	*CMSecurity*	2.10.5	*WhereIsMyWater*	1.9.3.86
8BallPool	3.5.2	*ZombieTsunami*	1.7.4	*SoundCloud*	2016.05.19	*Don'tTapTheWhiteTile*	4.0.3.5
Retrica	3.0.6	*DragRacing*	1.6.75	*SpotifyMusic*	5.4.0.858	*TriviaCrack*	2.12.0
FireFox	46.0.1	*BeautifulWidgets*	5.7.6	*PlantsVsZombies*	1.1.60	*CameraZOOMFX*	6.2.1
Netflix	4.5.1	*TuneinRadio*	15.5.1	*DoodleJump*	3.9.4	*NovaLauncher*	3.3

JavaScript code in this case is generated locally, we refer to this type of JavaScript usage as the local pattern (cf. Fig. 2a).

The local pattern can be implemented by using either of the following three WebView methods: `loadUrl(String url)`, `loadData(String data, String mimeType, String encoding)`, and `loadDataWithBaseURL(String data, String mimeType, String encoding, String historyUrl)`. For each method, there are two means to generate local JavaScript code: 1) add JavaScript in local HTML files located at "`file:///android_asset`" or "`file:///android_res`"; 2) embed scripts in the internal Java code via "`javascript:`". As an example, Fig. 4 shows these two means using `loadUrl()`.

3.1.2 File-Based Cross-Zone Vulnerability

The file-based cross-zone vulnerability will be introduced if file system access within WebView is allowed because SOP can be compromised through the following two means:

- The invocation of `setAllowFileAccessFromFileURLs(true)` allows local HTML to read local file contents via JavaScript. In this way, JavaScript in one file zone (local HTML) can access the contents of other file zones.
- The invocation of `setAllowUniversalAccessFromFileURLs(true)` allows local HTML to retrieve Web contents via JavaScript. However, JavaScript in one file zone can also access the contents of other http zones.

Note that prior to Android 4.2, the above two access permissions were enabled by default. To prevent this type of vulnerability, developers must explicitly disable these permissions.

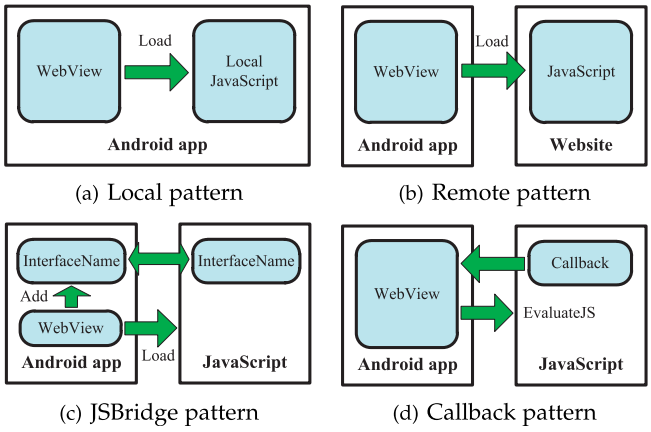


Fig. 2. The four JavaScript usage patterns.

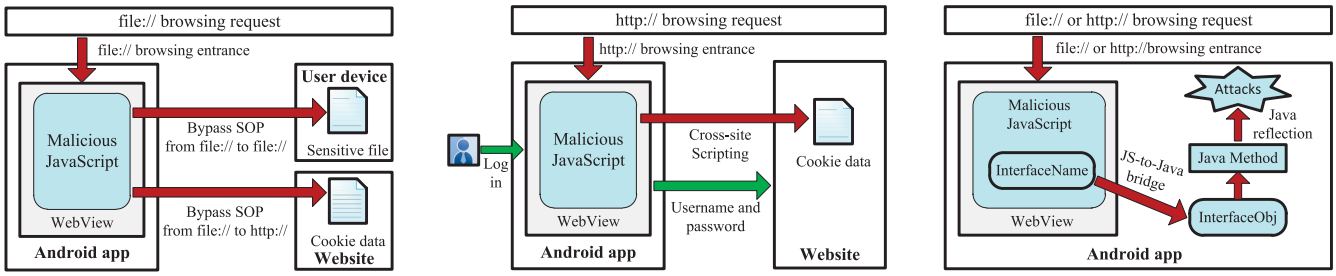
3.1.3 Attack Model

Fig. 3a illustrates the attack model for exploiting the file-based cross-zone vulnerabilities.

Attack Vector. If the manifest XML file of an app defines a component that can respond to a `file://` browsing request, then the attack vector is open. For example, in Fig. 5, since the activity component owns an *intent-filter*, which defines the action as "`VIEW`", the category as "`BROWSABLE`" or "`DEFAULT`", and the data as a "`file`" scheme ("`javascript`" scheme is similar), a malicious `file://` browsing request can be accepted by this intent-filter of the activity and delivered to WebView. Note that an intent is a messaging object that an app component uses to request an action from another component, and an intent-filter is a data structure associated with a component that specifies the type of intents that the component receives.

TABLE 3
How the Three JavaScript Usage Patterns can Lead to Their Related Vulnerabilities

Pattern	Vulnerability		Attack model		Victim version
	Root cause	Name	Attack vector	Attack means	
Local pattern	Inappropriate JavaScript zone access control	File-based cross-zone vulnerability	`file://` browsing entrance	Cross-zone scripting	All Android versions
Remote pattern	Navigable WebView	WebView UXSS vulnerability	`http://` browsing entrance	Cross-site scripting	Versions prior to Android 5.0
JSBridge pattern	Exposed Java object	JS-to-Java interface vulnerability	`file://` or `http://` browsing entrance	Java reflection abuse	Versions prior to Android 4.2



(a) File-based cross-zone vulnerability. (b) WebView UXSS vulnerability. (c) JS-to-Java vulnerability

Fig. 3. Attack models of the three JavaScript-related vulnerabilities.

Attack Methods. The SOP of a vulnerable app can be compromised through the following two means:

- SOP is compromised from `file://` to `file://`. For example, in Fig. 6a, the malicious JavaScript code (can be hidden in a local malware or a local third-party HTML file) sends an asynchronous request via the method `XMLHttpRequest()`[8] to access a sensitive file (`text.txt`) on the SD card of the user device. Subsequently, the contents of the file are displayed via the JavaScript function `alert()`.
- SOP is compromised from `file://` to `http://`. Fig. 6b shows an example that is similar to the example in Fig. 6a.

3.2 Remote Pattern and WebView UXSS Vulnerability

3.2.1 Remote Pattern

Developers often use WebView to open a remote website or a third-party JavaScript library. We refer to this type of JavaScript usage as the remote pattern (cf. Fig. 2b).

Similar to the local pattern, the remote pattern can be implemented by any of the first three WebView APIs in Table 1. Remote JavaScript codes can be loaded via either "`http://`" or "`https://`". Fig. 7 presents two examples of this pattern using `loadUrl()`.

3.2.2 WebView UXSS Vulnerability

The remote pattern may introduce WebView UXSS (universal cross-site scripting [7]) vulnerabilities. UXSS is a type of client-side loophole of browsers and browser extensions caused by failures in validating the user input. Attackers can inject malicious JavaScript to compromise SOP abided by browsers. This type of vulnerability is exploitable when WebView is navigable. When a user logs into a trusted website loaded by WebView, sensitive data, such as user name and password, are generally stored in cookies. Hence, these cookies could be stolen during page navigation if malicious JavaScript code is injected into the pages [2].

The Google Chrome browser based on the WebKit kernel was reported to have UXSS vulnerabilities [8], and thus, Android apps that use WebView of the WebKit kernel (prior to Android 4.4) are infected by this type of vulnerability. Starting with Android 4.4, WebView uses Blink, but still suffers from UXSS vulnerabilities until Android 5.0.

3.2.3 Attack Model

If a remote pattern is used in an app that exposes an attack vector, then the WebView UXSS vulnerability can be exploited (cf. Fig. 3b).

Attack Vector. If the manifest XML file defines a component that can respond to an `http://` browsing request, then the attack vector is open (cf. the example in Fig. 5).

```
myWebView.loadUrl("file:///android_asset/index.html");
myWebView.loadUrl("javascript:alert('Hello,JS')");
```

Fig. 4. Two examples of the local pattern.

```
<activity
    <intent-filter>
        <action android:name="android.intent.action.VIEW" >
        <category android:name="android.intent.category.
         DEFAULT">
        <category android:name="android.intent.category.
         BROWSABLE">
        <data android:scheme="file">
        <data android:scheme="http">
        <data android:scheme="https">
    </intent-filter>
</activity>
```

Fig. 5. An example of attack vectors.

8. XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest

```
<script>
    var aim = "file:///mnt/sdcard/test.txt";
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function(){
        if (xhr.readyState == 4){
        alert(xhr.responseText);
        }
    };
    xhr.open('GET',aim,true);
    xhr.send(null);
</script>
```

(a)

```
<script>
    var aim = "http://www.baidu.com";
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
    if (xhr.readyState == 4){
       alert(xhr.responseText);
    }
  };
  xhr.open('GET',aim,true);
  xhr.send(null);
</script>
```

(b)

Fig. 6. Two attack methods of the file-based cross-zone vulnerabilities.

```
myWebView.loadUrl("http://www.facebook.com/");
myWebView.loadUrl("http://code.jquery.com/jquery.js");
```

Fig. 7. Two examples of the remote pattern.

Attack Methods. There are various types of attack means, including script label, iframe label, *XMLHttpRequest* API, and so forth. Fig. 8 presents an example in which the JavaScript code opens a webpage via an iframe and creates a button in the main frame. Once the button is clicked, the JavaScript code can bypass the input checking via a crafted attribute containing a "\u0000" character and then access cookies of the page loaded within the iframe. This is caused by the inappropriate handling of the NULL byte (\u0000) by the url parser, where the data after "\u0000" are not checked. This attack requires that users have logged into the website.

Since UXSS vulnerabilities occur in the WebView itself, attackers can run malicious JavaScript code in any page loaded by the vulnerable WebViewto steal sensitive page contents, cookies, session tokens, and so forth. Stolen cookies may lead to serious attacks, such as session hijacking and user impersonating. Cross-site scripts can even rewrite the contents of HTML pages.

3.3 JSBridge Pattern and JS-to-Java Vulnerability

3.3.1 JSBridge Pattern

As mentioned in Section 2.2.3, Android provides a mechanism for apps to support the JavaScript-Java interaction. We refer to the JavaScript usage in this case as the interface pattern (cf. Fig. 2c).

In contrast to the first two patterns, the JSBridge pattern builds a bridge between apps and webpages. Through this bridge, apps can have control over Web contents via JavaScript. JavaScript code in webpages can also call the Java code. To use the JSBridge pattern, the method `addJavascriptInterface(Object object, String name)`

```
<iframe name="test" src="http://www.abc.com/">
</iframe>
<input type=button value="test"
onclick="window.open('\u0000javascript:alert(document.
        cookie)','test')" >
```

Fig. 8. An attack method of the WebView UXSS vulnerabilities.

```
myWebView.loadUrl("javascript:CheckUsername('"+strName+"
        ');");
myWebView.addJavascriptInterface(new MyJSInterface(),"JsBridge");
------
class MyJsInterface{
  void Prompt(){
      //prompt users
    }
}
```

(a)

```
<script>
function CheckUsername(username){
  if (!username.isValid()){
     JsBridge.Prompt();
   }
}
</script>
```

(b)

Fig. 9. An example of the JSBridge pattern: (a) Java code of an app, and (b) JavaScript code in webpages.

is invoked to inject a Java object as a global variable into the JavaScript environment. Thus, JavaScript code executed in WebView can call all public methods of the Java object via the provided name.

Fig. 9 shows an example that uses this pattern to achieve a form validation. The Java code in Fig. 9a creates a JS-to-Java interface object *JsBridge* and attaches it to the WebView. When WebView receives a form validation request from the user, it uses the method `loadUrl()` to call a JavaScript function `CheckUserName()` (cf. Fig. 9b). If the user name is invalid, the JavaScript function then invokes the Java method `Prompt()` through the *JsBridge* interface to prompt the user to enter a valid user name.

3.3.2 JS-to-Java Interface Vulnerability

The JSBridge pattern exposes a native JS-to-Java interface to the JavaScript environment, and it may suffer from JS-to-Java interface vulnerabilities if appropriate security countermeasures are not taken.

The interface name can easily be found via keyword searching in the decompiled app. Accordingly, malicious JavaScript code can call the method `getClass()` of the interface via the provided name and then abuse the Java reflection mechanism to obtain any other Java objects to invoke their methods. Although Google has deployed several security countermeasures in recent Android versions, the JS-to-Java interface vulnerabilities still exist in real-world apps:

1) Google requires that accessible methods of JS-to-Java interfaces must be explicitly annotated with `@JavascriptInterface` after Android 4.2. Since the Java reflection method `getClass()` is not annotated, reflection-based attacks can be reduced. However, this countermeasure only protects apps on Android 4.2 or later versions.
2) Google provides WebView with a method `removeJavascriptInterface(String name)` in all Android versions to remove previously injected

```
myWebView.addJavascriptInterface(new JSInterface(),"JsAttack");
```
(a)

```
<script>
function execute(){
    return JsAttack.getClass().forName("android.telephony.
        SmsManager")
  .getMethod("getDefault",null).invoke(null,null).sendTextMessage();
}
</script>
```
(b)

```
<script>
function execute(cmdArgs){
    return  JsAttack.getClass().forName("java.lang.Runtime").
        getMethod("getRuntime",null)
          .invoke(null,null).exec(cmdArgs);
}
function exec(){
    execute(["/system/bin/sh","-c","echo -n Attacks by JS
        >> /mnt/sdcard/file.txt"]);
}
</script>
```
(c)

Fig. 10. Two attack examples of JS-to-Java interface vulnerabilities: (a) Java code of an app, (b) one attack example, and (c) another attack example.

system-generated interfaces. However, this method is rarely used to remove app-defined interfaces in practice.

3) Google no longer allows the Java reflection method `getClass()` to be invoked via JS-to-Java interfaces after Android 6.0. However, it takes time for the new versions to spread, and many apps have backward compatibility with the older Android versions, which are still popular on user devices. According to Google,[9] as of July 2017, approximately 60 percent of Android devices are running versions prior to 6.0.

3.3.3 Attack Model

If the JSBridge pattern is used in an app that exposes an attack vector, then hackers can easily attack the app via injected malicious JavaScript code (cf. Fig. 3c).

Attack Vector. If the manifest XML file defines a component responsible for a `file://` or `http://` browsing request, then the attack vector is open (cf. the example in Fig. 5).

Attack Methods. Fig. 10 presents two attack examples of when an app exposes a JS-to-Java interface named *JsAttack* (cf. Fig. 10a):

- In Fig. 10b, the JavaScript code invokes the method `getClass()` of *JsAttack*, and it uses the Java reflection mechanism to obtain the Android *SmsManager* object. Then, the method `SendTextMessage()` of *SmsManager* is called to send a message. This attack would be successful if the app has permission to send SMS messages.
- In Fig. 10c, the JavaScript code uses a similar approach to obtain the Java *Runtime* object to execute any shell commands. The command in this example writes the text "`Attacks by JS`" into a file on the SD card. The attack would be successful if the app has the "write" permission to external storage.

Since malicious JavaScript code can use Java reflection to access the resources of both apps and user devices, these vulnerabilities can lead to data leaks. Although the attacks are restricted by permissions owned by the vulnerable apps, it is possible for attackers to run a root exploit for more permissions. More seriously, attackers may even exploit this type of vulnerability to install backdoors on user devices [9].

TABLE 4
Statistics of JavaScript Usage Patterns and Potential Vulnerabilities in the Top-100 Apps

Pattern	Total	Core	Lib	Potential vulnerability	Total	Core	Lib
Local	73	49	24	File-based	56	37	19
Remote	81	51	30	UXSS	79	52	27
Interface	72	50	22	JS-to-Java	69	51	18
Callback	63	43	20	-	-	-	-

4 Characterizing JavaScript Usage and Vulnerabilities in the Top-100 Apps

4.1 Usage of JavaScript

In this section, we report on the JavaScript usage in the top-100 Android apps:

> *Finding 1.* Among the top-100 apps, 89 enable JavaScript execution. JavaScript is mainly used in four scenarios. Approximately 1/3 of the usage is attributed to third-party libraries.
> *Implications.* JavaScript is widely used in Android apps. Third-party libraries occupy a large proportion of JavaScript usage.

According to our empirical study, JavaScript is mainly used in the following scenarios:

1) An app uses JavaScript to validate form elements before they are submitted to the server. For example, *Dropbox* uses JavaScript to check the strength of a password by client-side JavaScript.
2) Webpages leverage JavaScript to interact with users. For example, *Facebook* uses WebView to display webpages that use JavaScript for user interaction.
3) JavaScript manipulates the elements of HTML pages. For example, *Gmail* uses JavaScript to format messages.
4) Third-party libraries require JavaScript support. For example, in addition to loading advertisements, the app *Subway Surfers* employs the *Flurry* framework, which uses JavaScript for data statistics.

As summarized in Table 4, nearly 2/3 of the JavaScript usage is due to the core functionality of the apps themselves. The remaining 1/3 is caused by third-party libraries, such as frameworks, advertisements, and so forth. Frameworks facilitate the use of some integrated functions, which accelerates app development. Although advertisements increase the revenue for the app authors, these third-party libraries may also introduce JavaScript-related vulnerabilities. Moreover, many libraries are rarely or slowly updated [10] but are constantly used in new versions of apps, which increases the exposure to vulnerabilities. To this end, we also study

9. https://developer.android.com/about/dashboards/index.html

TABLE 5
The Number of Apps in Which Each JavaScript Usage Pattern is Employed in Each Third-Party Library

Lib \ Ptn.	Local	Remote	Interface	Callback
Jirbo	13	11	2	12
Mopub	9	11	3	0
Vungle	8	6	0	0
Flurry	6	4	1	0
Chartboost	6	3	0	0
Tapjoy	5	4	7	5
Millennial	4	3	2	3
Appboy	3	3	0	0
InMobi	1	7	7	5
Mraid	2	2	2	3

TABLE 6
The Number of Apps in Which Each Type of Potential JavaScript Vulnerability is Due to Each Third-Party Library

Lib \ Vulner.	File-based	UXSS	JS-to-Java
Jirbo	12	11	2
Mopub	6	11	3
Vungle	5	6	0
Flurry	0	4	1
Chartboost	5	3	0
Tapjoy	5	4	7
Millennial	3	3	2
Appboy	2	3	0
InMobi	0	7	7
Mraid	1	2	2

the JavaScript usage patterns and vulnerabilities in 10 of the most popular third-party libraries used in these apps. Tables 5 and 6 summarize the results. Overall, all these libraries use the local and remote patterns frequently, most of them use the JSBridge pattern, and half of them use the callback pattern. Moreover, all of them suffer from at least two types of vulnerabilities.

4.2 Pattern Usage and Vulnerabilities

Table 7 summarizes the results of our empirical study of the top-100 apps with respect to JavaScript usage patterns, potential vulnerabilities and attacks. Fig. 11 illustrates the minimum, maximum, and mean occurrences of each pattern in an app. Note that the results here are based on our manual inspection. More details are described in the following.

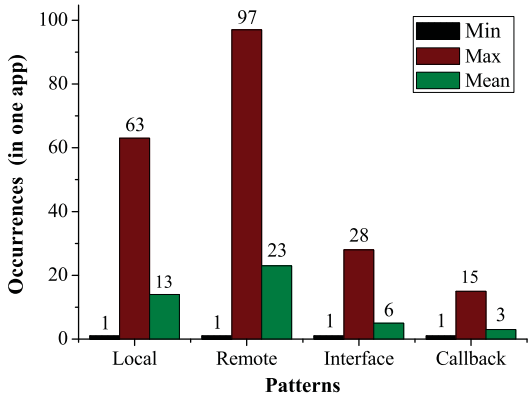


Fig. 11. Occurrences of each pattern in an app.

4.2.1 Usage of Each Pattern

In this part, we report on the usage of each pattern in all 89 JavaScript-enabled apps.

Finding 2. The local pattern is used by 73 (82.02 percent) apps. For each app, the pattern appears 1 to 63 times, with 13 and 14 times as the medians. Among these 73 apps, 24 (32.88 percent) employ third-party libraries.
Implications. The local pattern is used both widely and frequently in apps.

Finding 3. The remote pattern is used by 81 (91.01 percent) apps. For each app, the pattern appears 1 to 97 times, with 17 times as the median. Among these 81 apps, 30 (37 percent) employ third-party libraries.
Implications. The remote pattern is the most commonly used pattern in apps. It is also the most frequently used pattern in an app.

Finding 4. The JSBridge pattern is used by 72 (80.90 percent) apps. For each app, the pattern appears 1 to 28 times, with two as the median. Among these 72 apps, 22 (30.56 percent) employ third-party libraries.
Implications. The JSBridge pattern is widely used in apps due to its powerful functionality to implement the interaction between webpages and apps. The usage frequency of the JSBridge pattern in an app is relatively low.

4.2.2 Exposure to Each Vulnerability

In this part, we report on the app vulnerabilities related to JavaScript usage.

TABLE 7
JavaScript Usage Patterns and Vulnerabilities in the Top-100 Apps

Pattern		Potential vulnerability			Attack vectors		Victims
Name	Total	Name	No	Yes	Closed	Open	
Local	73	File-based	17	56	49	7	7
Remote	81	UXSS	2	79	39	40	40
Interface	72	JS-to-Java	3	69	34	35	35
Callback	63	-	-	-	-	-	-

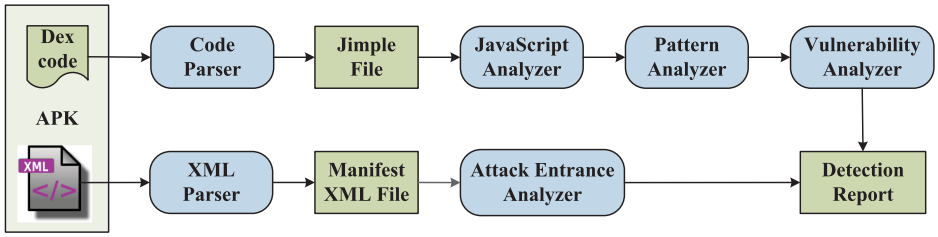


Fig. 12. Overall architecture of JSDroid.

Finding 5. Among all 73 apps using the local pattern, 56 (76.7 percent) set inappropriate control of JavaScript access to zones, and only 7 (12.5 percent) of these 56 expose the attack vectors. Among the 56 apps, 19 (33.9 percent) employ at least one third-party library.
Implications. When the local pattern is used, JavaScript access to zones is often allowed, but the attack vector is closed by most apps. Only less than 10 percent are exploitable. Moreover, approximately 1/3 of file-based cross-zone vulnerabilities are attributed to third-party libraries.

Finding 6. Among all 81 apps using the remote pattern, 79 (97.5 percent) apps can navigate webpages within WebView. Among the 79 apps, 40 (50.6 percent) expose the attack vectors, and 27 (34.18 percent) employ at least one third-party library.
Implications. When the remote pattern is used, WebView is always navigable, and the attack vector is often exposed. This indicates that most developers are unaware of this type of vulnerability. Moreover, approximately 1/3 of WebView UXSS vulnerabilities are due to third-party libraries.

Finding 7. Among all 72 apps using the JSBridge pattern, 69 (95.8 percent) expose JS-to-Java interfaces. Among the 69 apps, 35 (50.7 percent) expose the attack vectors, and 18 (26.09 percent) employ at least one third-party library.
Implications. When the JSBridge pattern is used, JS-to-Java interfaces are often exposed. This indicates that the majority of developers fail to implement security countermeasures when using the JSBridge pattern. Moreover, approximately 1/3 of JS-to-Java interface vulnerabilities are caused by third-party libraries.

5 Evaluation on 1,000 Real-World Apps

To extensively study JavaScript-related vulnerabilities in the Android market, we developed a static analysis tool called JSDroid and have applied it on 1,000 large real-world Android apps to automatically detect their potential JavaScript-related vulnerabilities and to study the performance of our tool. These 1,000 apps are randomly selected from the Android market, each with at least 50,000 downloads (per specific version counted by APK4Fun: https://www.apk4fun.com/) and 350K lines of code and 9.7M Dex file on average. We downloaded their most recent version as of August 2016 from APK4Fun.

In this section, we first present the design and implementation of JSDroid followed by our experimental results. All experiments are performed on a computer with an Intel Core i7 3.6 GHz CPU with 16 GB of memory and running Windows 7 and JDK 1.7.

5.1 The JSDroid Tool

We design JSDroid based on the characteristics of JavaScript usage patterns and vulnerabilities described in Section 3 and implement it based on Soot [11][10] and AXMLPrinter.[11] Fig. 12 depicts the architecture of JSDroid. The input is a list of APK files, and the output is a report that records the detected patterns and vulnerabilities. For each app, JSDroid performs a linear scan of the code and resources with the following six modules:

Code Parser. In an Android APK, there is a code file in Dalvik Executable (DEX) format. Since DEX is a binary encoded file, it is difficult to apply program analysis on it. To this end, the code parser converts the Dalvik bytecode to Jimple (Soot's principal intermediate representation) [12] files with the help of Soot. Jimple has a clear code structure that facilitates the static analysis.

XML Parser. The manifest XML file of an Android APK is difficult to read because Android developers often use code obfuscation techniques to protect their apps. The XML parser converts the obfuscated manifest XML file to an easy-to-read XML file with the help of AXMLPrinter.

JavaScript Usage Analyzer. Jimple summarizes 15 types of Java statements, such as *AssignStmt*, *InvokeStmt*, *IfStmt*, and so forth. The JavaScript usage analyzer scans all Jimple files. If it finds one *InvokeStmt* that invokes the method `setJavaScriptEnabled()` of WebSettings with the operand "1" (Java boolean value true), then the app uses JavaScript.

Pattern Analyzer. The pattern analyzer scans all Jimple files to find *InvokeStmts* that invoke APIs related to each of the four JavaScript usage patterns. A challenge in this module is to differentiate the local pattern and the remote pattern because they both use the same WebView APIs (`loadUrl()`, `loadData()`, and `loadDataWithBaseURL()`) but with different parameter values. To address this problem, we analyze the value of the string URLs passed to each API in the following four ways:

1) If the URL is a constant string and the string prefix is "`file:///`", "`javascript:`" or "`<script>`", then we classify it as the local pattern; otherwise, we classify it as the remote pattern (the prefix could be "`http://`" or "`https://`").

10. Soot-GitHub. http://github.com/Sable/soot
11. AXMLPrinter. http://code.google.com/archive/p/android4me-/downloads

TABLE 8
Experimental Results on 1,000 Large Real-World Apps

Total	Use JS	Pattern		Potential vulnerability		Attack vector		Victim
		Name	No	Name	No	Closed	Open	
1,000	806	Local	796	File-based	702	628	74	74
		Remote	762	UXSS	708	516	192	192
		Interface	618	JS-to-Java	600	415	185	185
		Callback	393	-	-	-	-	-

2) If the string URL is a static class member, then we analyze its definition in the class and classify it by following (1).
3) If the URL is a local variable of *StringBuffer* or *StringBuilder*, then we construct a *UnitGraph* (control flow graph implemented in Soot) for the current method and conduct data-flow analysis. If a constant string flows to the variable, we follow (1) to classify the pattern. In this case, we also analyze popular *StringBuffer*/*StringBuilder* functions such as `toString()`, `append()` and `init()` and process them according to their semantics.
4) If the URL is a variable passed from the method parameter or returned from a method call, then we use the call graph to find the corresponding caller or callee methods and again analyze the values in those methods. The call graph is constructed based on the soot-infoflow-android framework as a part of FlowDroid [13].

Vulnerability Analyzer. After a pattern is identified, the vulnerability analyzer determines whether the specific use of this pattern leads to the related vulnerability according to the precondition of each type of vulnerability (cf. Section 3):

1) *Detection of file-based cross-zone vulnerabilities in local pattern*. We traverse all statements in the current method (which contains an *InvokeStmt* comprising the local pattern) and in the current class. If there are three *InvokeStmts* that respectively invoke the three WebSettings APIs (`setAllowFileAccess()`, `setAllowFileAccessFromFileURLs()`, and `setAllowUniversalAccessFromFileURLs()`) and if the values of their parameters are all false (indicating that cross-zone access is strictly forbidden), then we classify that the app does not suffer from file-based cross-zone vulnerabilities; otherwise, it may suffer from this type of vulnerability.
2) *Detection of WebView UXSS vulnerabilities in remote pattern*. We traverse all statements in the current WebView class and determine its navigability by the following rules. If the WebView does not have a WebViewClient, then it is non-navigable. Otherwise, we continue to analyze each subclass of the WebViewClient. If the subclass does not overwrite the method `shouldOverrideUrlLoading()`, or in the overwritten method, it overloads a new URL or returns false, then the current WebView is navigable. If the WebView is navigable, then the app may suffer from WebView UXSS vulnerabilities; otherwise, it does not.
3) *Detection of JS-to-Java interface vulnerabilities in JSBridge pattern*. Our analysis has two steps. First, we check whether the app limits the usage of the JSBridge pattern to Android 4.2 or later versions. Second, we check whether the app removes a previously used JS-to-Java interface (i.e., injected Java object from a WebView). If both of these conditions are false, then the app may suffer from JS-to-Java interface vulnerabilities. For the first condition, we perform a dominator analysis on the current *InvokeStmt* (which satisfies the JSBridge pattern) and check whether it is dominated by an *IfStmt* with the condition "Build.VERSIONS.SDK_INT $>=$ 17" (here, 17 corresponds to the version number of Android 4.2). We first identify whether such an *IfStmt* exists; if yes, we then check whether the *IfStmt* dominates the current *InvokeStmt* or the method containing it. For the second condition, we traverse all methods in the current class and check whether the WebView API `removeJavascriptInterface()` is invoked. If yes, we remove that interface from the maintained list of JSBridge patterns identified in the app. After all JSBridge patterns are analyzed, if the list is empty, then the app does not have JS-to-Java interface vulnerabilities.

Attack Vector Analyzer. After one vulnerability is identified, the attack vector analyzer checks whether the app exposes an attack vector by tracking the manifest XML file. This module scans the manifest file from outer labels to inner labels iteratively to find a `file://` browsing entrance or an `http://` browsing entrance. For instance, if it finds one *intent-filter* label that owns the action label "`VIEW`", the category label "`BROWSABLE`" or "`DEFAULT`", and the data label "`file`", then the app exposes a `file://` browsing entrance.

5.2 Experimental Results

Table 8 summarizes the experimental results of the 1,000 large real-world Android apps based on JSDroid.

JavaScript Usage. JavaScript is prevalently used in these 1,000 apps as 806 of them enable JavaScript execution. The local pattern is used by 796 apps, the remote pattern is used by 762 apps, and the JSBridge pattern is used by 618 apps.

Vulnerability Exposure. In total, JSDroid finds that 201 of the 1,000 apps involve exploitable JavaScript-related vulnerabilities (i.e., the attack vectors are open), with a total of 451 exploitable vulnerabilities (many apps have more than one type of vulnerability). Among the 796 apps using the local pattern, 702 apps set inappropriate control of JavaScript access to zones, and thus, these apps have potential file-based cross-zone vulnerabilities. JSDroid reports that 74 of these apps open the attack vectors and can thus be attacked. Among the 762 apps using the remote pattern, 708 apps set

TABLE 9
Effectiveness Evaluation of JSDroid

App name	Data origin	Pattern times				Vulner. numbers				Criteria			
		P1	P2	P3	P4	V1	V2	V3	TP	FP	FN	precision	recall
Opera Mini	Empirical study	38	13	7	0	0	0	5	61	1	2	98.4%	96.8%
	JSDroid	39	11	7	0	0	0	5					
ES File Explorer	Empirical study	31	38	8	3	31	14	7	128	2	4	98.5%	97.0%
	JSDroid	32	34	8	4	32	14	7					
Line	Empirical study	12	13	1	4	0	11	1	42	0	0	100%	100%
	JSDroid	12	13	1	4	0	11	1					
Tango	Empirical study	13	20	4	3	13	4	3	59	2	2	96.7%	96.7%
	JSDroid	14	18	4	3	14	4	3					
Viber	Empirical study	10	12	2	4	0	1	2	31	0	0	100%	100%
	JSDroid	10	12	2	4	0	1	2					
Dropbox	Empirical study	14	9	6	1	14	2	6	51	1	1	98.1%	98.1%
	JSDroid	14	8	6	2	14	2	6					
Go Launcher EX	Empirical study	21	19	7	0	21	4	5	77	0	0	100%	100%
	JSDroid	21	19	7	0	21	4	5					
Google TTS	Empirical study	4	6	0	2	4	1	0	17	1	0	94.4%	100%
	JSDroid	4	6	0	3	4	1	0					
Pics Art Photo	Empirical study	17	25	7	3	0	4	5	61	4	0	93.8%	100%
	JSDroid	18	25	7	4	0	6	5					
Chat On	Empirical study	13	27	2	0	13	4	3	61	1	1	98.4%	98.4%
	JSDroid	13	28	1	0	13	4	3					
Average	-	-	-	-	-	-	-	-	-	-	-	97.8%	98.7%

WebView as navigable and are potentially vulnerable to WebView UXSS vulnerabilities. JSDroid finds that 192 of the apps expose the attack vectors, which can be attacked. Among the 618 apps using the JSBridge pattern, 600 apps expose JavaScript interfaces in versions prior to Android 4.2 and thus suffer from potential JS-to-Java interface vulnerabilities. JSDroid finds that 185 of them open the attack vectors and can thus be attacked.

5.2.1 *Performance of* JSDroid

In this section, we report the performance of our tool JSDroid in terms of effectiveness and efficiency.

We apply JSDroid to the top-100 apps and compare the results with the manual results obtained by our empirical study on these apps. Table 9 summarizes the experimental results on 10 apps randomly selected from the top-100 apps. P1, P2, P3, and P4 represent local pattern, remote pattern, JSBridge pattern, and callback pattern, and V1, V2, and V3 represent file-based cross-zone vulnerabilities, WebView UXSS vulnerabilities, and JS-to-Java interface vulnerabilities, respectively. We list the number of patterns and vulnerabilities identified by JSDroid and our empirical study. The results of our empirical study are used as the ground truth, which was established by the authors of this paper. In fact, since the conditions to determine whether an app involves JavaScript-related vulnerabilities is clear, the validity of the manual check is not a severe problem.

Overall, JSDroid generates very few false positives/negatives in identifying JavaScript usage patterns and vulnerabilities, and the average precision and recall are 97.8 and 98.7 percent, respectively. The high precision and recall give credit to Soot and the problem we study. With Soot, JSDroid transforms the APK files into Jimple code and then scans the Jimple code to check how the apps use JavaScript, whether vulnerabilities exist, and whether the attack vectors are open. Since the major task of the scanning is to search relevant APIs in the Jimple code and the arguments of the APIs are not complex, the accuracy of JSDroid is guaranteed. One reason for the false positive/negative is that in our empirical study, we employ Dex2Jar to decompile APK files into readable pure-text Java source code, whereas JSDroid uses Soot to decompile APK files into code-structured Jimple files. The code of some apps obtained by these two different tools (i.e., Dex2Jar and Soot) are not completely consistent, which results in the deviation of the results returned by JSDroid.

For time efficiency, JSDroid completes a batch of vulnerability detection on 100 apps in less than 20 minutes. In other words, JSDroid needs only 12 seconds on average to analyze a complex app.

5.2.2 *Comparison to Existing Tools*

Although several studies on JavaScript-related vulnerabilities in Android apps exist [2], [3], [4], there is no available tool that we can directly compare with. Chin and Wagner [3] claim that they built a tool for detecting JS-to-Java interface vulnerabilities, but neither the tool nor the experimental results based on the tool are available. Wu et al. [4] designed

TABLE 10
Experimental Comparison Between JSDroid and FileCross

Package name	FileCross	JSDroid		
	V1	V1	V2	V3
com.airwatch.browser	×	×	√	√
com.apps4mm.browserformm	√	√	√	×
com.bluecoat.k9.android	×	√	√	×
com.browser.sogood.ui	×	×	√	×
com.candy.browser	×	×	√	√
com.cloudmosa.puffinFree	×	×	√	√
com.compal.android.browser	√	√	√	×
com.crowbar.beaverlite	×	×	×	√
com.ddm.netviewer	√	√	√	√
com.droidboosters.tigerbrowser	×	√	√	×
com.lpedroid.internet	×	×	×	×
com.mahoni.browser	×	×	×	×
com.MobileWebSite.v1	×	×	√	×
com.mx.browser.appendix	×	×	√	√
com.ninesky.browser	√	√	√	√
com.opera.mini.android	×	×	√	√
com.ploxpex.browserexample	×	×	×	×
com.soshall.apps.browser	√	√	×	×
com.unibera.privatebrowser	×	×	×	×
com.websearch.browser	√	√	√	×
com.whattheapps.fbrowsers	×	×	×	×
com.wSuperFastInternetBrowser	×	×	×	×
harley.browsers	√	√	×	×
iron.web.jalepano.browser	√	√	√	√
it.nikodroid.offline	×	×	×	×
jp.ddo.pigsty.HabitBrowser	×	√	√	×
me.android.browser	×	×	×	×
mobi.mgeek.TunnyBrowser	×	√	√	√
net.caffeinelab.pbb	×	×	×	×
steffen.basicbrowserfree	√	√	√	√

a system (called FileCross) for detecting file-based cross-zone vulnerabilities. Although the tool is unavailable, the detailed experimental results on 30 Android apps are publicly available. We hence apply JSDroid to the same set of 30 apps to compare it with FileCross. All APK files of these apps are downloaded from the links provided in their paper.

Table 10 summarizes the results, where V1, V2, and V3 represent file-based cross-zone vulnerabilities, WebView UXSS vulnerabilities, and JS-to-Java interface vulnerabilities, respectively. A tick (√) in the table indicates that a tool detects a certain type of vulnerability in an app; a cross (×) indicates that a tool does not find the existence of a certain type of vulnerability. Specifically,

1) FileCross can only detect file-based cross-zone vulnerabilities in an app, whereas JSDroid can detect all three types of JavaScript-related vulnerabilities.
2) All file-based cross-zone vulnerabilities identified by FileCross can also be detected by JSDroid, whereas FileCross fails to find this type of vulnerability in some apps. For instance, JSDroid identifies file-based cross-zone vulnerabilities in four apps whose package names are `com.bluecoat.k9.android`, `com.droidboosters.tigerbrowser`, `jp.ddo.pigsty.HabitBrowser`, and `mobi.mgeek.TunnyBrowser`, respectively, while FileCross fails to do this. We successfully conduct real attacks on these four apps, which confirms these vulnerabilities.
3) Moreover, FileCross can only detect file-based cross-zone vulnerabilities in Android browsers, whereas JSDroid is applicable to all types of Android apps.

6 Exploiting Vulnerabilities in Real Apps

For vulnerable apps with attack vectors, we are able to conduct real attacks by exploiting the vulnerabilities. In this section, we present our results of vulnerability exploitation in 30 real-world apps (15 from the top-100 apps and 15 from the 1,000 large apps). All 451 of the vulnerabilities identified by JSDroid are exploitable. Due to time and resource limitations, we only inspected the exploitable vulnerabilities in 30 apps.

Table 11 summarizes our attack results on these 30 apps. For each app, we list our attack results for the three types of JavaScript-related vulnerabilities launched on three different Android versions, including 4.1.3, 4.3.1, and 5.0. A tick (√) implies a successful attack, whereas a cross (×) means a failed attack. A hyphen (-) indicates that the app cannot be attacked because it is free of the corresponding vulnerabilities or no attack vector is exposed. We next present three case studies to show the successful exploitation of each of the three types of vulnerabilities.

Boat Browser is a popular lightweight mobile browser whose 8.7.4 version has over 55,000 downloads. JSDroid finds that this version includes all three types of JavaScript-related vulnerabilities. Here, we only describe how to exploit file-based cross-zone vulnerabilities to attack this app. The app opens a `file://` browsing entrance for users to search for resources on websites. As depicted in Fig. 13a, we input the path of a local HTML file from the entrance, and we use the app to open our specially designed page. The malicious JavaScript code in our page is executed and performs several cross-zone scripting operations that successfully steal contents in a text file. As depicted in Fig. 13b, the content "`abcdefg`" in the file is displayed.

Sohu News is a top-10 news reading app in China. Its 4.0 version has over 200,000,000 downloads. This version involves WebView UXSS vulnerabilities, and can be attacked on Android versions prior to 5.0. The app opens an `http://` browsing entrance for users to add links to their comments. As depicted in Fig. 14a, we input a URL from the entrance and use the app to open our designed page. The malicious JavaScript code on our page uses an iframe to open the Baidu home page and to conduct a series of cross-site operations that successfully obtain user's cookies on the iframe page. Fig. 14b shows that the user's cookie in Baidu is stolen.

ES File Explorer is a full-featured resource manager in Android. Its 4.0.5 version has over 3,000,000 downloads. JSDroid finds that this version involves all three JavaScript-related vulnerabilities. Herein, we only show its JS-to-Java interface vulnerabilities. The app opens both `file://` and `http://` browsing entrances for users to search for resources in local file systems and websites. As shown in Fig. 15a, we input a URL from the entrance, and we use the app to open our designed page exploiting WebView UXSS vulnerabilities. The malicious JavaScript code on this page traverses the JavaScript context iteratively to find the objects (JS-to-Java interface objects) that own the `getClass()` method. If such an object is found, then the JavaScript code outputs its name in this page and invokes the Java reflection

TABLE 11
Results of Vulnerability Exploitation in 30 Real-World Apps

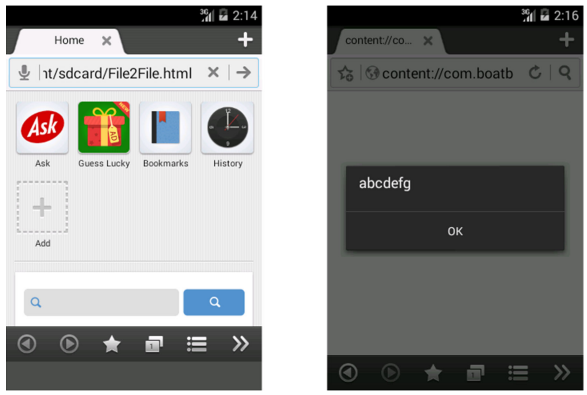
App	File-based cross-zone vulner.				WebView UXSS vulner.				JS-to-Java interface vulner.			
	Attack?	4.1.3	4.3.1	5.0	Attack?	4.1.3	4.3.1	5.0	Attack?	4.1.3	4.3.1	5.0
Amazon Kindle	Yes	√	√	√	No	-	-	-	No	-	-	-
Sogou Explorer	Yes	√	√	√	No	-	-	-	No	-	-	-
Microsoft Office	Yes	√	√	√	No	-	-	-	No	-	-	-
Firefox	Yes	√	√	√	Yes	√	√	×	No	-	-	-
Facebook	No	-	-	-	Yes	√	√	×	No	-	-	-
Wechat	No	-	-	-	Yes	√	√	×	No	-	-	-
QQ Browser	No	-	-	-	Yes	√	√	×	No	-	-	-
Instagram	No	-	-	-	Yes	√	√	×	No	-	-	-
Sina Viber	No	-	-	-	Yes	√	√	×	No	-	-	-
Dropbox	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Line	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Shazam	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Viber	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Pics Art Photo	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Kik	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Opera Mini	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Sohu News	No	-	-	-	Yes	√	√	×	Yes	√	×	×
UC Browser	No	-	-	-	Yes	√	√	×	Yes	√	×	×
Amap	No	-	-	-	Yes	√	√	×	Yes	√	×	×
eBay	No	-	-	-	No	-	-	-	Yes	√	×	×
Adobe Reader	No	-	-	-	No	-	-	-	Yes	√	×	×
Baidu HD	No	-	-	-	No	-	-	-	Yes	√	×	×
Snow Ball	No	-	-	-	No	-	-	-	Yes	√	×	×
Sogou Input	No	-	-	-	No	-	-	-	Yes	√	×	×
Apc Browser	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×
Boat Browser	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×
Es File Explorer	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×
Hao123 Browser	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×
MX Player	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×
MX Browser	Yes	√	√	√	Yes	√	√	×	Yes	√	×	×

mechanism to write a text file onto the users' SD card. As presented in Fig. 15b, the app exposes five JS-to-Java interface objects. As shown in Fig. 15c, we successfully write five text files named by these objects.

7 Discussion

Note that even when users upgrade their device/Android versions, the JavaScript-related vulnerabilities are still severe in practice. The reasons are as follows. Our vulnerability result demonstrates that the file-based cross-zone vulnerabilities affect all Android versions. In January of 2108, HUAWEI reported that nearly 25 percent of the top-1,000 Android apps developed by Chinese companies suffer from file-based cross-zone vulnerabilities.[12] Although the WebView UXSS vulnerabilities affect versions prior to Android 5.0 and the JS-to-Java interface vulnerabilities affect versions prior to Android 4.2, the adoption of new Android versions typically takes a long time, and a huge number of apps are still running on older versions of Android [14]. According to Google,[13,14] there are 2 billion monthly active devices on Android, and close to 10 percent are running Android 4.3 or lower versions and more than 4 percent are running Android 4.1 or lower versions. This indicates that the file-based cross-zone vulnerabilities may affect 2 billion users, and the WebView UXSS and JS-to-Java interface vulnerabilities may still affect hundreds of millions of users.

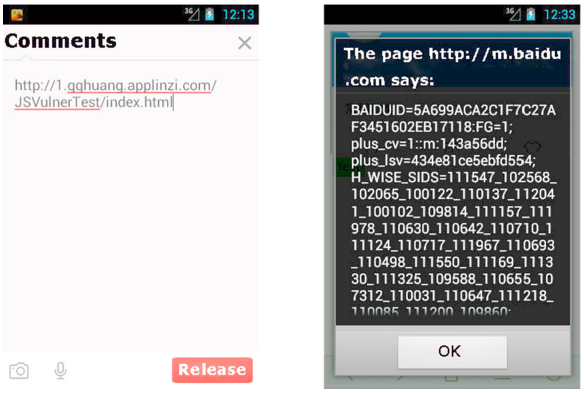
Our result clearly highlights the need for Android developers to close the attack vectors by specifying more strict permissions in the app manifest file and for Android users to upgrade to newer Android versions. Detailed suggestions are as follows:



(a) (b)

Fig. 13. Attack case 1 (*Boat browser* 8.7.4; file-based cross-zone vulnerabilities; Android 5.0).

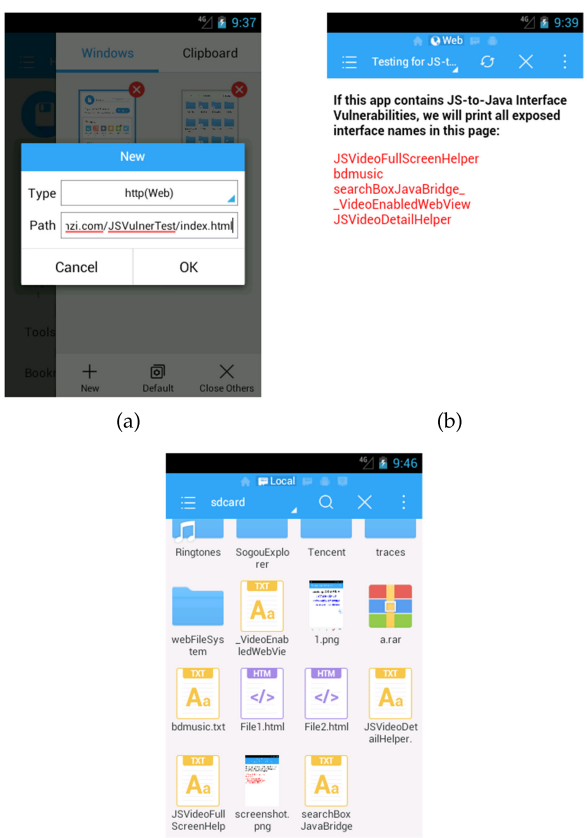
12. https://testerhome.com/topics/11750, in Chinese, accessed in 03/2018.
13. https://developer.android.com/about/dashboards/index.html, accessed in 07/2017.
14. https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users, accessed in 07/2017.



(a) (b)

Fig. 14. Attack case 2 (*Sohu news* 4.0; WebView UXSS vulnerabilities; Android 4.3.1).

- *Limit JavaScript execution in file://URLs.* We find that many apps allow WebView to execute JavaScript regardless of different protocol contents loaded by WebView. In fact, it is unnecessary to support JavaScript for `file://` protocol contents because apps often use this protocol to display local HTML files.
- *Restrict WebView navigability.* We find that many apps fail to correctly limit the navigability of WebView. Developers often use a WebViewClient to control the navigability of a WebViewbut unintentionally set WebView as navigable. A better approach is to set WebView as navigable in the method



(a) (b)

(c)

Fig. 15. Attack case 3 (*ES file explorer* 4.0.5; JS-to-Java interface vulnerabilities; Android 4.1.3).

```
myWebView.setWebViewClient(new WebViewClient()){
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url
        ){
      if (URI.parse(url).getHost().equals("www.mydomain.com")){
        view.loadUrl(url);
        return false;
      }
        Intent intent=new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse(url));
        startActivity(intent);
        return true;
    }
});
```

Fig. 16. Example of restricting WebView navigability.

`shouldOverrideUrlLoading()` only if the new URL is in the same domain as the origin, as depicted in Fig. 16.

- *Do not expose available JS-to-Java interfaces.* It is best to use the JSBridge pattern only in Android versions higher than 4.2. For instance, `Facebook`, `Instagram`, and `Amazon Kindle`all use this approach to avoid JS-to-Java interface vulnerabilities. If the JSBridge pattern is required for an Android version earlier than 4.2, then developers should remove the registered Java objects after their use.
- *Sanitize user inputs.* A typical reason for successful JavaScript injection is that developers fail to check and filter user inputs. General input data of apps are treated as pure texts, whereas URL data must be sanitized to filter keywords of malicious JavaScript, such as "`script`", "`iframe`", "`XMLHttpRequest`" and "`getClass`".
- *Limit the attack vector.* Blocking the `file://` and `http://` browsing entrances can help effectively prevent attacks. For example, *Gmail* and *Google Text-to-Speech* use this approach to block the attack even though their apps involve vulnerabilities.
- *Update apps and devices.* New versions of apps often repair vulnerabilities. For instance, the latest version of *Adobe Reader* repairs JS-to-Java interface vulnerabilities in old versions. Moreover, higher versions of Android devices also protect users' privacy. As shown in Table 11, some vulnerabilities do not exist in devices with Android versions higher than 4.4.
- *Online Vulnerability Testing.* We also develop an online JavaScript vulnerability testing service (publicly available at http://bit.ly/JSVulTest) for users to test their apps. According to the attack methods of each vulnerability, we implement this service by injecting vulnerability-exploiting JavaScript code into our specially designed webpages. When apps open these webpages, the JavaScript code will be executed to detect the three types of JavaScript-related vulnerabilities and will inform users of the detected vulnerabilities.

8 Related Work

JavaScript-related vulnerabilities in Android have received increasing attention in recent years [2], [3], [4], [15], [16], [17]. Our work is distinguished by presenting a systematic study on JavaScript usage in real-world popular Android

apps and their relationships with the three classes of JavaScript-related vulnerabilities. We review existing studies according to the type of vulnerabilities they focus on.

File-Based Cross-Zone Vulnerabilities. The work by Chin and Wagner [3] is among the first to study this type of vulnerability. The work by Wu and Chang [4] studies four types of attacks that exploit `file://` vulnerabilities. However, none of them conduct in-depth research on this type of vulnerability, e.g., they fail to analyze the root cause and build the attack model, and they only cover a small number of real-world apps.

WebView UXSS Vulnerabilities. The work by Bhavani [2] is among the first to study cross-site scripting attacks on Android WebView. This work analyzes Web-based APIs of HttpClient to model potential attacks, but unfortunately, no concrete attack model is given. A number of studies exist on cross-site scripting vulnerabilities in Web applications [18], [19], [20], [21], [22], as well as several approaches for mitigating cross-site scripting attacks [15], [16]. These studies only discuss the impact of these vulnerabilities in general Web applications, without analyzing them within the context of Android apps.

JS-to-Java Interface Vulnerabilities. Luo et al. [17] conduct a small-scale manual investigation and discuss two attack means of this type of vulnerability, including 1) malicious webpages attacking apps, and 2) malicious apps attacking webpages. However, attacks related to new WebView APIs are not studied. Moreover, they do not perform any real attack. Chin and Wagner [3] extend Luo's work by identifying excess authorization vulnerabilities, which is named the JS-to-Java interface vulnerability in our work. Neugschwandtner et al. [23] present some case studies on the vulnerabilities and note that developing a static analysis tool to detect the vulnerabilities is rewarding. Thomas et al. [14] propose an exponential decay model for Android API vulnerabilities, which predicts that the fix of JS-to-Java interface vulnerability will not have been deployed to 95 percent of devices until the end of 2017.

Some works study JavaScript-related vulnerabilities in hybrid Web/mobile application frameworks [24], [25], [26]. Yu and Yamauchi [24] first identify security-sensitive APIs and then detect threats at runtime. If threats are found, the users are notified of the threats. The work by Xing et al. [25] finds code injection attacks caused by cross-site scripting in HTML5-based apps, particularly *PhoneGap* apps.[15] These studies inspire our work to analyze Android hybrid frameworks and develop a static analysis tool for detecting potential JavaScript-related vulnerabilities in apps.

9 Conclusions

We have presented an empirical study on the top-100 most-popular real-world Android apps demonstrating that JavaScript is widely used but that such usage may lead to various types of security threats. We identify four JavaScript usage patterns and three types of related vulnerabilities. For each type of vulnerability, we present a root cause analysis and build the attack model. We have also developed a static analysis tool named JSDroid that can automatically detect JavaScript-related vulnerabilities in Android apps. We have applied JSDroid to 1,000 large real-world Android apps, and we found hundreds of exploitable vulnerabilities. Moreover, we have successfully launched real attacks on 30 of these apps.

15. https://phonegap.com

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant No. 2017YFB1001801, the National Natural Science Foundation of China under Grant No. 61761136003, and the Natural Science Foundation of Jiangsu Province under Grant No. BK20171427.

References

[1] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 250–261.

[2] A. B. Bhavani, "Cross-site scripting attacks on Android WebView," *Int. J. Comput. Sci. Netw.*, vol. 2, no. 2, pp. 1–5, Apr. 2013.

[3] E. Chin and D. Wagner, "Bifocals: Analyzing WebView vulnerabilities in Android applications," in *Proc. 14th Int. Workshop Inf. Secur. Appl.*, 2013, pp. 138–159.

[4] D. Wu and R. K. C. Chang, "Analyzing Android browser apps for file: // vulnerabilities," in *Proc. 17th Int. Conf. Inf. Secur.*, 2014, pp. 345–363.

[5] C. Karlof, U. Shankar, J. D. Tygar, and D. A. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2007, pp. 58–71.

[6] G. Yang, J. Huang, and G. Gu, "Automated generation of event-oriented exploits in android hybrid apps," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[7] US-CERT, "Cve-2015–1275," 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015–1275

[8] UXSS vulnerability in chrome for android, 2012. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=144813

[9] T. R. Team, "Alert: Android webview addjavascriptinterface code execution vulnerability," 2013. [Online]. Available: http://blog.trustlook.com/2013/09/04

[10] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 356–367.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proc. Conf. Centre Adv. Studies Collaborative Res.*, 1999, Art. no. 13.

[12] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 27–38.

[13] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel , "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2014, pp. 259–269.

[14] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android API vulnerabilities: Case study on the Javascript-to-Java interface," in *Proc. 23rd Int. Workshop Secur. Protocols XXIII*, 2015, pp. 126–138.

[15] I. Yusof and A. K. Pathan, "Mitigating cross-site scripting attacks with a content security policy," *IEEE Comput.*, vol. 49, no. 3, pp. 56–63, Mar. 2016.

[16] A. Javed, J. Riemer, and J. Schwenk, "SIACHEN: A fine-grained policy language for the mitigation of cross-site scripting attacks," in *Proc. 17th Int. Conf. Inf. Secur.*, 2014, pp. 515–528.

[17] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, 2011, pp. 343–352.

[18] G. A. D. Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana, "Identifying cross site scripting vulnerabilities in web applications," in *Proc. 6th Int. Workshop Web Site Evol. Testing*, 2004, pp. 71–80.

[19] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 171–180.
[20] M. K. Gupta, M. C. Govil, G. Singh, and P. Sharma, "XSSDM: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications," in *Proc. Int. Conf. Adv. Comput. Commun. Inf.*, 2015, pp. 2010–2015.
[21] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns, "From Facepalm to brain bender: Exploring client-side cross-site scripting," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1419–1430.
[22] L. A. Meyerovich and V. B. Livshits, "Conscript: Specifying and enforcing fine-grained security policies for Javascript in the browser," in *Proc. 31st IEEE Symp. Secur. Privacy*, 2010, pp. 481–496.
[23] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: WebView exploitation," in *Proc. 6th USENIX Workshop Large-Scale Exploits Emergent Threats*, 2013, pp. 1–4.
[24] J. Yu and T. Yamauchi, "Access control to prevent malicious Javascript code exploiting vulnerabilities of WebView in Android OS," *IEICE Trans. Inf. Syst.*, vol. 98-D, no. 4, pp. 807–811, Apr. 2015.
[25] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. Scottsdale*, 2014, pp. 66–77.
[26] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *Proc. 21st Annu. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–15.

Wei Song received the PhD degree from Nanjing University, China, in 2010. He is currently an associate professor in the School of Computer Science and Engineering, Nanjing University of Science and Technology, China, and was a visiting scholar at Technische Universität München, Germany. His research interests include software engineering and methodology, services and cloud computing, Android application analysis, and process mining. He was invited to a Schloss Dagstuhl Seminar held in August, 2016. He has published in premiere computer science journals such as *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, and *IEEE Transactions on Software Engineering*. He is a member of the IEEE.

Qingqing Huang received the ME degree from the Nanjing University of Science and Technology, China, in 2017. She is now a software engineer. Her research interests include software engineering, program analysis, security, and JavaScript.

Jeff Huang received the PhD degree from the Hong Kong University of Science and Technology, in 2012 and was a postdoc with the University of Illinois at Urbana-Champaign from 2013 to 2014. He is an assistant professor with Texas A&M University. His research focuses on developing practical techniques and tools for improving software reliability and performance. He has published extensively in premiere software engineering conferences and journals such as PLDI, OOPSLA, ICSE, FSE, ISSTA, the *ACM Transactions on Software Engineering and Methodology*, the *IEEE Transactions on Software Engineering*. He is a member of the IEEE.