Indirect File Leaks in Mobile Applications

Daoyuan Wu and Rocky K. C. Chang
Department of Computing
The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
{csdwu, csrchang}@comp.polyu.edu.hk

***Abstract*—Today, much of our sensitive information is stored inside mobile applications (apps), such as the browsing histories and chatting logs. To safeguard these privacy files, modern mobile systems, notably Android and iOS, use sandboxes to isolate apps' file zones from one another. However, we show in this paper that these private files can still be leaked by *indirectly* exploiting components that are trusted by the victim apps. In particular, we devise new *indirect file leak* (IFL) attacks that exploit browser interfaces, command interpreters, and embedded app servers to leak data from very popular apps, such as Evernote and QQ. Unlike the previous attacks, we demonstrate that these IFLs can affect both Android and iOS. Moreover, our IFL methods allow an adversary to launch the attacks remotely, without implanting malicious apps in victim's smartphones. We finally compare the impacts of four different types of IFL attacks on Android and iOS, and propose several mitigation methods.**

I. Introduction

Mobile applications (apps) are gaining significant popularity in today's mobile cloud computing era [3], [4]. Much sensitive user information is now stored inside mobile apps (on mobile devices), such as Facebook authentication tokens, Chrome browsing histories, and Whatsapp chatting logs. To safeguard these privacy files, modern mobile systems, notably Android and iOS, use sandboxes to isolate apps' file zones from one another.

However, it is still possible for an adversary to steal private app files in an *indirect* manner by exploiting components that are trusted by the victim apps. We refer to this class of attacks as *indirect file leaks* (IFLs). Fig. 1 illustrates a high-level IFL model. Initially, an adversary cannot directly access a private file, formulated as $a \not\Leftarrow s$. If the adversary can send crafted inputs to a deputy[1] inside the victim app ($a \rightarrow d$) and these inputs can trigger the deputy to send the private file to the adversary ($d \rightarrow s \Rightarrow a$), then the adversary can indirectly steal the private file. The whole process, $a \rightarrow d \rightarrow s \Rightarrow a$, achieves the goal of $s \Rightarrow a$, causing an IFL.

In this paper, we devise new IFL attacks that exploit browser interfaces, command interpreters, and embedded app servers to leak files from very popular apps, such as Evernote and Tencent QQ. Unlike prior works [51], [47] that only show local IFL attacks on Android, we demonstrate that three out of our four IFL attacks affect both Android and iOS. We summarize these attacks below.

- *`sopIFL` attacks* bypass the same-origin policy (SOP), which is enforced to protect resources originating from different *origins* (i.e., the combination of scheme, domain, and port), to steal private files via browsing interface deputies. Although our prior work [47] has demonstrated such attacks on Android by exploiting numerous Android browsers' SOP flaws on the `file://` scheme, we are extending it in this paper to a number of vulnerable iOS apps, such as the very popular Evernote, Tencent QQ, and Mail.Ru. We also confirm that the latest iOS 8 fails to enforce the appropriate SOP on `file://`. In our analysis, the root case of this attack is that the legacy web SOP is found to be inadequate for the local schemes, such as `file://`. Eradicating the problem may call for an enhanced SOP.
- *`aimIFL` attacks* also leverage browsing interfaces as deputies, but they do not need to violate SOP. It can do so by injecting and executing unauthorized JavaScripts *directly* on target files, instead of requiring a malicious file to bypass SOP to access target files as in the `sopIFL` attacks. Popular Android browsers, such as Baidu, Yandex, and 360 Browser, can be easily compromised in this way, allowing their private files (e.g., cookie and browsing history) to be stolen. The high-profile 360 Mobile Safe and Baidu Search are also exploitable. Besides these Android apps, we further uncover a vulnerable iOS app, myVault.
- *`cmdIFL` attacks* exploit command interpreters as deputies inside victim apps to execute unauthorized commands for file leaks. We demonstrate that the top command apps on Android, Terminal Emulator and SSHDroid, can be stealthily exploited to execute arbitrary commands, possibly with the root privilege. This will jeopardize their own files (e.g., command histories and

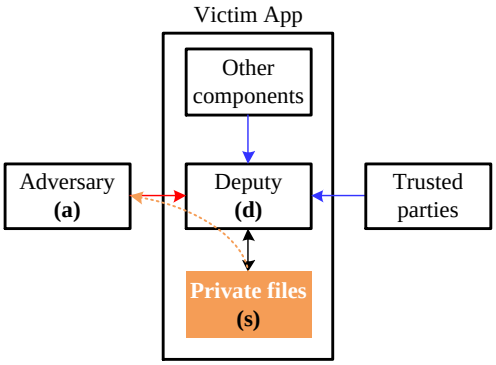


Fig. 1. A high-level IFL model.

[1]We borrow this term from the classic confused deputy problem [31] to represent a trusted component in victim apps.

private configuration files), sensitive user photos stored on SD cards, and even other apps' private files.

- *`serverIFL` attacks* send unauthorized file extraction requests to embedded app server deputies inside victim apps to obtain private files. All of the tested popular server apps, such as WiFi File Transfer (Android) and Simple Transfer (iOS), are vulnerable to these attacks. It is worth noting that both the `cmdIFL` and `serverIFL` attacks use previously unexplored deputies—command interpreters and embedded app servers—to launch IFL attacks for the first time in mobile platforms.

Besides the cross-platform vulnerability, our IFL methods also allow an adversary to launch the attacks remotely, without implanting malicious apps in victim's phones as in prior works [51], [47]. These IFL attacks can be launched both locally (in the same phone) and remotely (in an Intranet/Internet). Table I highlights the identified IFL vulnerabilities and their major attack channels. Besides local IFL attacks, we show that browsers, such as Baidu and Yandex Browser, can be remotely exploited by enticing the victim to access a web page. Email apps (e.g., Mail.Ru) and social apps (e.g., QQ) can be similarly compromised if the victim opens a malicious attachment or file transmitted by a remote adversary. In other remote attacks, the adversary can scan the whole Intranet, locate open ports, and exploit vulnerable server apps installed on the victim phone.

Furthermore, we analyze the differences between Android and iOS in terms of the impact of the IFL attacks. These differences are caused by different system architectures and app design practices between Android and iOS. Our analysis shows that a common iOS app practice could lead to more powerful and pervasive `sopIFL` attacks on iOS than Android. On the other hand, three iOS system characteristics help lessen the impacts on iOS for the other three IFL attacks. These findings can help developers and OS providers build more secure apps and mobile systems.

Ethical considerations. All of our vulnerability testing is conducted using our own mobile devices and test accounts. The tests *never* affect the data security of real-world users. As the IFL attacks in mobile apps are client-side vulnerabilities, they *cannot* affect the server-side integrity.

Real-world impacts. We have reported most of the identified vulnerabilities to their vendors in a responsible way and are in the process of reporting the remaining vulnerabilities. All of the contacted vendors have acknowledged our reports. For example, Evernote has listed us in its security hall of fame. Baidu has ranked one of our reports as the most valuable vulnerability report of the second quarter of 2014, and Qihoo 360 has issued us the highest award in its mobile bug bounty program history. We have also offered our suggestions to the vendors to fix the identified vulnerabilities.

Contributions. To summarize, we make the following three contributions:

- We devise four new IFL attacks that, for the first time, can affect both Android and iOS and are exploitable not only locally but also remotely. (Section III & IV)
- We identify a number of zero-day IFL vulnerabilities in popular Android and iOS apps and uncover a serious SOP issue in the latest iOS 8 system. (Section IV)
- We analyze the differences between Android and iOS in terms of the IFL attacks' impacts and propose several methods to mitigate the attacks. (Section V & VI)

TABLE I
FOUR IFL VULNERABILITIES AND THEIR ATTACK CHANNELS.

	Attack Channels	
	Remote	Local
`sopIFL`	Evernote, Tencent QQ Mail.Ru	UC & QQ browsers 360 & Mail.Ru Cloud
`aimIFL`	Baidu, Yandex, and 360 browsers 360 Mobile Safe, Baidu Search, myVault	
`cmdIFL`	SSHDroid	Terminal Emulator
`serverIFL`	WiFi File Transfer, Simple Transfer	

II. BACKGROUND

A. Sandbox-based App Isolation

Both Android and iOS use sandbox-based app isolation to build a trustworthy mobile environment. Each app resides in its own sandbox, with its code and data isolated from other apps. This isolation is usually enforced at the kernel level. For example, Android uses UNIX UID-based protection to isolate each app, in which each app is treated as an independent user and runs in a separate process with a unique `uid`.

Each app's sensitive files are stored in their own system-provided isolated (or private) file zone. Unless an app actively leaks a file (i.e., a direct file leak), other apps have no access to the protected files. The widely deployed SEAndroid MAC system [41] further thwarts the risks incurred by direct file leaks. However, IFLs can still occur in the presence of both sandbox-based isolation and MAC. Although the actual executer of the file access is the legal victim app, the file request is actually initiated and crafted by an adversary. Encryption-based defenses, such as encrypting all private app files, face similar limitations as the MAC. To sum up, IFL remains a serious, and yet unsolved, threat, which motivates our study.

B. Terminology

In Table II, we summarize the terms used throughout this paper.

TABLE II
TERMS AND THEIR DESCRIPTIONS.

Term	Description
Private files	The files stored in apps' isolated file zones. In nonlocal IFL attacks, they also include files on a SD card, e.g., user photos.
Target file	A private file the adversary wants to steal in an IFL attack.
Permission	A form of privilege representation. For example, the `INTERNET` permission on Android and the `Contact` permission on iOS.
Root	A superuser privilege. For example, a *rooted phone* is a phone that enables superuser privilege for apps.
Browsing interface	Or browsing component. A component with browsing capability, usually built with Android's WebView or iOS's UIWebView.
Command interpreter	An app component that can interpret and execute commands.
App server	A server component embedded in an app.

III. THE IFL ATTACKS

In this section, we first describe the adversary model and then detail the four types of IFL attacks introduced in Section I.

A. Adversary Model

We consider the following three types of adversaries in our IFL attacks. A local adversary can launch only local attacks, whereas the Intranet and Internet adversaries *remote* IFL attacks.

- A *local* adversary is an attack app installed on the same smartphone as the victim app. It requires few or no permissions and does not exhibit any typical malicious app behavior [50]. The root privilege is *never* used by this attack app. We also do not consider screenshot attacks [34] that require strong assumptions.
- An *Intranet* adversary resides in the same Intranet as the victim's mobile device. It can send network requests to any other node within the Intranet. It can sniff the nearby wireless traffic and retrieve unencrypted content. We do not assume that it can launch effective ARP spoofing attacks, as network administrators can detect such anomalous events.
- An *Internet* adversary can be located in any host in the Internet. It remotely compromises a victim by (i) enticing a victim to browse a web page under the adversary's control, and/or (ii) sending the victim a malicious file via email, chatting app, social network, and other means.

B. Bypassing SOP on Browsing Interfaces

The `sopIFL` attacks bypass SOP to steal private files via browsing interfaces. The deputy in this attack is the browsing interface or rendering engine, whose SOP enforcement is flawed and cannot prevent malicious JavaScript codes from accessing a private file. All apps that contain browsing components are potential victims.

The `file://` scheme is an ideal medium to launch the `sopIFL` attacks. Two parts of SOP enforcement on `file://` can be exploited to steal local private files. The adversary can cross the origin from a web domain to access local file content, if the cross-scheme SOP enforcement for `http(s)://` to `file://` (labelled as `SOPf1`) is broken. Alternatively, the adversary can leverage a local malicious HTML file in one path to steal a target file in another path. In this case, the `file://` SOP enforcement between the two file origins (labelled as `SOPf2`) must be bypassed. Since failure of enforcing `SOPf1` is rare in modern rendering engines, we focus on the `sopIFL` attacks that will bypass the `SOPf2`.

Our recent study [47] shows that Android does not effectively enforce `SOPf2`. However, little is known about iOS. This is where our contribution for the `sopIFL` attacks lies. Contrary to our expectation, these attacks can have higher impact on the iOS ecosystem than the Android's. Our testing using iPhone 6 reveals that even the latest iOS 8 does not properly enforce `SOPf2`. Indeed, iOS *never* guarantees this policy (Section IV-A). In Section V, we identify a common practice among the iOS apps that could lead to more pervasive and powerful `sopIFL` attacks on iOS than Android.

We believe that the root problem is that the legacy SOP cannot adequately cover the local schemes, such as `file://`. The typical web SOP principle (i.e., the legacy SOP) allows file A (at `file:///dir1/a.html`) to access file B (at `file:///dir2/b.txt`) because the two origins share the same scheme, domain (i.e., 127.0.0.1 or localhost), and port. In practice, this legal behavior fails to meet the security requirements for `file://`, especially in the mobile environment. Therefore, an enhanced SOP for local schemes, such as adding the "path" element to the current three-element SOP tuples, is needed for eradicating this vulnerability. We reported our iOS findings to Apple on 19 January 2015 and suggested them to use an enhanced `file://` SOP at the system or engine level.

C. Unauthorized JavaScript Execution on Target Files

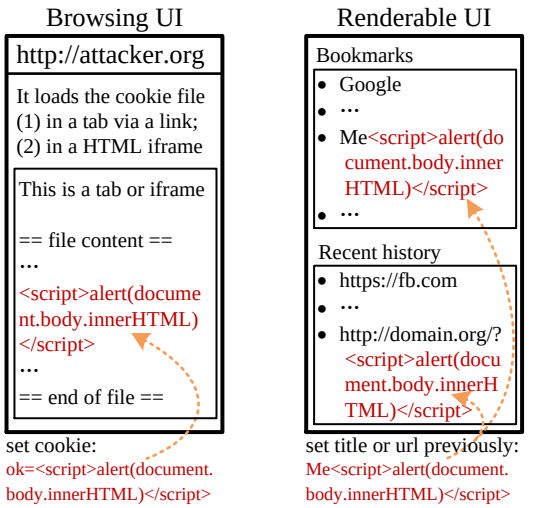
The `aimIFL` attacks could be regarded as an advanced variant of the `sopIFL` attacks. Both attacks use the browsing interface as the deputy, but the `aimIFL` attack does not violate SOP. It can do so by injecting and executing unauthorized JavaScripts *directly* on target files, instead of requiring a malicious file to bypass SOP to access target files as in the `sopIFL` attacks.

The `aimIFL` attacks usually consist of two steps. Unauthorized JavaScript codes are first *injected* into the target file. This can be achieved in different ways, depending on the type of the target files. For example, if the target is a cookie file, the JavaScript codes can be injected via a website's cookie field. The target files are then *loaded* and rendered. The previously injected JavaScript is executed to steal the current file content via an HTML document object model variable, such as `document.body.innerHTML`. As JavaScript only accesses the current document, this attack does not violate SOP, thus setting this attack apart from the `sopIFL` attack.

It is worth noting that the two steps can also be performed simultaneously without the user's knowledge. A victim user's private files will be stolen when he browses a web page under the attacker's control. In this paper, we focus on designing remote `aimIFL` attacks, although they can also be conducted locally.

We identify two types of remote `aimIFL` attacks illustrated in Fig. 2. In the first type, a web page tries to load a target file through local schemes like `file://` and `content://`. The file can be loaded by a file link or an HTML `iframe`. The link-based loading requires an extra user clicking, whereas the iframe-based loading is automatic and does not require any user action. Before loading the target cookie file, the web page injects malicious JavaScript codes (e.g., `<script>alert(document.body.innerHTML) </script>`) via the web cookie field. Once the cookie file is successfully loaded, the JavaScript inside it can steal the cookie content. As will be shown in Section IV-B, the popular 360 Mobile Safe, Baidu, and Yandex browsers are all vulnerable to this type of `aimIFL` attack.

The second type of the `aimIFL` attacks does not actively load and render target files. Instead, it exploits victim apps' ability to load the content of target files into a renderable user interface (UI), such as those containing WebView widgets. For example, some browser apps load browsing histories



(a) Attacks that actively load a target file.

(b) Attacks that exploit victim apps' file loading features.

Fig. 2. Remote aimIFL attacks.

from the history file into a renderable UI. Using this app-loading feature, an adversary simply injects the JavaScript into the target file. For example, as illustrated in Fig. 2(b), the adversary injects an unauthorized JavaScript into the history file through the title or URL field of a web page. When the user switches to the renderable UI, a new history log containing the malicious title or URL is displayed. The embedded JavaScript is then executed to steal the history file. These passively loaded files are usually rendered in the local domain (e.g., under `file://`). The adversary can also combine an `aimIFL` attack with a `sopIFL` attack to steal other private files that are not loaded by the victim apps. We will detail the affected apps in Section IV-B.

D. Unauthorized Command Execution on Command Interpreters

The `cmdIFL` attacks exploit command interpreters (as deputies) inside victim apps to execute file operation related commands for IFL attacks. We consider explicitly embedded command interpreters, such as those in command terminal apps. In other words, (remote) code execution vulnerabilities contained in host apps are out of the scope.

An app is vulnerable to a `cmdIFL` attack only when it can be injected with unauthorized commands. To leak private files, the injected commands can (i) set a world-readable file permission by invoking the `chmod` command, (ii) export a file to a public SD card via the `cp` command, or (iii) send a file to a remote server through commands like `scp`. If the victim app has root permissions, all of these commands can be used to steal private files in other, possibly more sensitive, apps.

To discover and exploit a `cmdIFL` vulnerability, an adversary needs to identify channels that can be used to confuse command interpreters to accept unauthorized commands. The cross-app component communication channel [25], [23] on Android can be used to launch local `cmdIFL` attacks, if the command interpreter components are exposed. An adversary can also exploit the URL scheme [44] to achieve the same, if not better, attack impacts. Moreover, general remote `cmdIFL` attacks are also possible, if the command interpreter accepts remote command requests. In addition, victim apps' configurations stored in public storage can be changed to indirectly inject commands. Accessibility services can also be misused to mimic user commands [32].

A `cmdIFL` attack could also be launched through a GUI to attack file manager apps. An adversary can force these apps to perform unauthorized UI-based file operations to leak file contents. However, this is not a real threat, because such file operations have to be conducted by victim users and thus are easily noticed. A smarter adversary can wait for users to open a sensitive file and launch screenshot attacks [34] to sniff the content. We exclude this scenario also from our treat model, because this requires the adversary to closely monitor the victim's activity.

E. Unauthorized File Extraction via Embedded App Servers

The confused deputies here are the app servers embedded in victim apps. An adversary sends unauthorized file extraction requests to exploit vulnerable embedded app servers for obtaining private files. The affected app servers are mainly file servers (e.g., those that support `http://` and `ftp://` requests) that provide users with file transmission service between phones and desktops. The command servers mentioned in the last section are another type of candidate servers. Some apps that support multi-function servers are also affected, such as the very popular AirDroid app, which has over 10 million installs.

A `serverIFL` attack can be conducted in three ways. First, a local attack can be launched from another app installed on the same smartphone as the victim app. It scans the local hosts' ports and sends packets to the port listened to by the victim app. Second, adversarial nodes (e.g., phones or laptops) in the same Intranet can attack the victim app in another device. Within the same Intranet, it is also easy for an adversary to identify a port opened by the victim app. Third, we show that remote `serverIFL` attacks are also possible. Attack vectors are delivered through the victim's desktop browsers when they browse a malicious web page.

A successful `serverIFL` attack may need to bypass authentication set up to protect the victim app, such as an authentication code (e.g., user password) or a confirmation action (e.g., clicking a confirmation button). Our evaluation in Section IV-D, however, shows that the current authentication in server apps is nearly broken. Some apps use no or weak authentication (e.g., using only four random numbers). Almost all channels for transmitting authentication codes and subsequent session tokens are unencrypted. An Intranet adversary can easily sniff these secrets to bypass the authentication step.

IV. Uncovering IFL Vulnerabilities in Android and iOS Apps

A. SopIFL Vulnerabilities

We uncover a number of vulnerable iOS apps summarized in Table III. We evaluate them using an iPhone 6 (with the

TABLE III
iOS APPS VULNERABLE TO THE sopIFL ATTACKS.

Category	Vulnerable Apps	Attack Channel
Browser	UC, Mercury Baidu, Sogou, QQ browsers	Local
Cloud Drive	Mail.Ru Cloud Baidu Cloud, 360 Cloud	Local & Web
Note/Read	Evernote, QQ Reader	Local & Web
Email	Mail.Ru	Remote
Social	Tencent QQ	Remote
Utility	Foxit Reader, OliveOffice	Local

latest iOS 8) and an iPad 3 (with iOS 7). Our evaluation shows that both iOS 7 and 8—which are used in around 95 percent of all iOS devices [1]—do not enforce SOPf2 at the engine level. Since all iOS apps can use only the default web engine, the current solution for mitigating the sopIFL vulnerabilities can only be done on the application level.

To launch a sopIFL attack on the iOS platform, a "malicious" HTML file must be delivered to the victim app. We have identified three such channels.

Local The adversary can design stealthy iOS apps (e.g., the Jekyll app [45]) to launch local sopIFL attacks, because some iOS apps accept external HTML files from other apps through the "open with" feature[2]. Similarly, Android browsers often use exposed browsing interfaces [47]. We notice that local attacks can also be conducted "remotely" by leveraging other apps' remote channels. For example, an adversary first sends an HTML file to the popular WeChat app installed on a victim device. Since WeChat will not open this type of file, it will ask the user to open the file using another app. We find that browser and cloud drive apps are likely to be affected by these local attacks.

Web An adversary can deliver attack vectors through the web service interfaces of mobile apps. For example, cloud drive and note apps support file sharing on the web. An adversary can share an HTML file with a victim via web interfaces.

Remote Remote sopIFL attacks are possible for some iOS apps. The attachment mechanism in email apps is an ideal channel to launch targeted remote sopIFL attacks. For example, once a Mail.Ru iOS user opens an attached HTML file from an adversary email, the adversary can steal the victim's private Mail.Ru files remotely. Similarly, the file sending mechanism in social apps, such as the popular Tencent QQ, can also be exploited.

Besides HTML files, we anticipate other types of files for launching sopIFL attacks. For example, the commonly used PDF and flash file formats can execute embedded JavaScript codes in a desktop environment [5], [17]. The current mobile systems have limited support for flash and only run basic JavaScript in PDF files. Once these systems are improved, we expect that these new attack vectors will bypass the existing protection. For example, WeChat is immune to the current sopIFL attacks by disabling the opening of HTML files. However, it allows opening PDF files to be opened, which could be exploited for future sopIFL attacks.

[2]Two "open with" demos implemented in Dropbox and WeChat are available at http://goo.gl/H7KXeM.

TABLE IV
APPS VULNERABLE TO THE aimIFL ATTACKS.

Attack Name	Vulnerable Apps
aimIFL-1 via file://	Baidu Browser, On The Road
aimIFL-1 via content://	360 Mobile Safe
aimIFL-1 via intent://	Yandex and 360 browsers Baidu Search, Baidu Browser
aimIFL-2 on Android	org.easyweb.browser Internet Browser, Smart Browser Shady Browser, Zirco Browser
aimIFL-2 on iOS	myVault

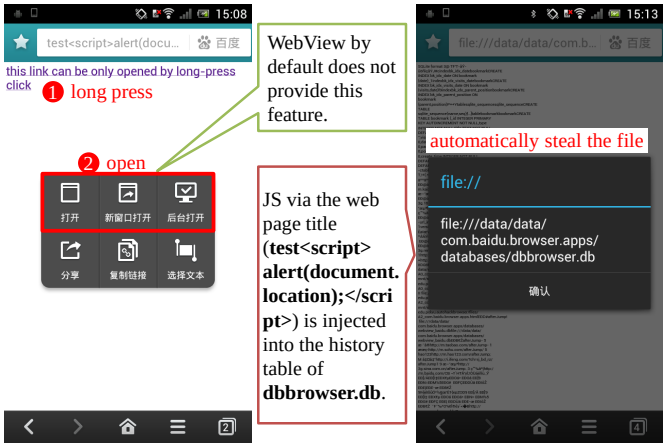


Fig. 3. An aimIFL-1 attack exploiting Baidu Browser.

B. AimIFL Vulnerabilities

We summarize the aimIFL vulnerabilities in Table IV. As discussed in Section III-C, these vulnerabilities can be classified into two types: aimIFL-1 and aimIFL-2.

aimIFL-1 attacks. We attempt an aimIFL-1 attack via file:// on two Android apps, Baidu Browser and On The Road. We find it difficult to directly load a file:// content (e.g., via an HTML iframe) from a web page on Android. We thus use an alternative method that asks users to click a file:// link embedded in a web page. This method is able to exploit On The Road, as the app renders the file:// link clickable. However, similar to desktop browsers, file:// links in Baidu Browser are not clickable. We find that an adversary can entice a victim to long-press the link, allowing Baidu Browser to pop up a dialog that the user can click. The target file is then rendered and its contents are stolen automatically. The attack procedure is illustrated in Fig. 3.

We find the content:// scheme on Android can also be exploited by aimIFL-1 attacks. This scheme is used to retrieve content or data from the corresponding content provider components. Surprisingly, we find that a web page can load a local file via the content:// scheme if the associated content provider implements the openFile(Uri uri, String mode) API. Launching aimIFL-1 attacks via content:// is therefore even easier than file://. We have successfully launched this attack to remotely exploit 360 Mobile Safe. Because of the seriousness of this exploit, 360 has issued us the highest award in its mobile bug bounty program history.

Terada [43] points out that the intent:// scheme can be used to remotely attack local Android components. Following this idea, we independently identify several popular browsers and the Baidu Search app that can be exploited by aimIFL-1 attacks via intent://. These victim apps satisfy the following three conditions, which make them exploitable.

- They contain Intent.parseUri() to intercept an intent:// URI and generate an Intent structure. This Intent can invoke any component of the victim, even a private component that has not been exposed to other apps. An adversary can thus design a crafted intent:// URI to deliver attack vectors to a target component.
- They include a component that imports external Intent parameters to WebView.loadUrl(String url). An adversary can thus control this component to render an arbitrary URI.
- The victim component in the last step allows file:// access and its JavaScript execution. The victim can therefore render a target file via file:// and execute its embedded JavaScript codes.

All the aimIFL-1 attacks discussed above affect only Android apps. We will explain why it is hard to launch aimIFL-1 attacks on iOS in Section V.

aimIFL-2 attacks. Launching the aimIFL-2 attacks successfully requires two conditions. The victim app must be able to load the content of a target file into a WebView-based UI gadget, and the adversary must be able to inject an unauthorized JavaScript into the target file.

We use the first condition to facilitate the search for vulnerabilities. We focus on Android browsers and iOS WebView-based apps in [9] and inspect their screenshots to choose which apps to install and test. The WebView.loadDataWithBaseURL API is useful for locating vulnerable Android browsers, because developers often invoke this API to load the local content and their associated assets. We use this API and its first parameter value (e.g., starting with "file://") to search Android browser codes and identify a vulnerable open source Zirco browser. Interestingly, this vulnerable Zirco design is also used in several other browsers, including "Browser for Android" (org.easyweb.browser) that affects around one million users.

One iOS app, myVault, is exploitable by the aimIFL-2 attack. This app allows users to store their private photos, bookmarks, and passwords. Its bookmark store page is an entry point where a "malicious" bookmark can be injected to steal the victim's bookmarks. Even worse, as iOS does not enforce SOP well on file://, an adversary can therefore steal other sensitive content through a crafted bookmark.

C. CmdIFL Vulnerabilities

Table V lists the identified cmdIFL vulnerabilities. We select command terminal and server apps in Google Play, because these apps are more likely to contain command interpreters than normal apps. More specifically, we evaluate the top apps, Terminal Emulator and SSHDroid, as they are the most likely to be installed by users with these requirements. For example, Terminal Emulator for Android has over 10 million installs, whereas the top two terminal apps have less than 0.5 million installs.

TABLE V
CMDIFL VULNERABILITIES.

Apps	Vulnerability Cause	Attack Channel	# of Installs
Terminal Emulator	The command component is exposed.	Local	10M+
SSHDroid	The command server is weakly protected.	Local & Intranet	500K+

```
<activity-alias android:name="RunScript"
        android:targetActivity="RemoteInterface"
        android:permission="jackpal.androidterm.permission.RUN_SCRIPT">
    <intent-filter>
        <action android:name="jackpal.androidterm.RUN_SCRIPT" /> (1)
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity-alias>
<permission android:name="jackpal.androidterm.permission.RUN_SCRIPT"
        android:protectionLevel="dangerous" ... >

<activity android:name="RemoteInterface"
        android:excludeFromRecents="true">
    <intent-filter>                                                  (2)
        <action a:name="jackpal.androidterm.OPEN_NEW_WINDOW"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

Fig. 4. Manifest excerpt of Terminal Emulator.

Both tested command apps are found to be vulnerable to the cmdIFL attacks. A local attack app can execute arbitrary commands using the identity of Terminal Emulator, such as exporting its private files (e.g., command histories and private configuration files) to a public SD card. The root cause of this vulnerability is that the exposed terminal command component can be invoked by other local apps with arbitrary command parameters. Interestingly, we find that Terminal Emulator tries to protect its command component with a dangerous-level permission, as shown in part (1) of Fig. 4. The rationale behind this design is that all command invocations (via the jackpal.androidterm.RUN_SCRIPT action) now have the appropriate authorization through the permission mechanism. Unfortunately, the adversary does not need to touch the protected command proxy component ("RunScript"). Instead, it directly invokes the underlying command component ("RemoteInterface" in part (2) of Fig. 4), which is by default exposed by Android's intent filter mechanism [23]. Consequently, a crafted input can force the "RemoteInterface" component to execute commands. We reported this issue and its demo attack code to Terminal Emulator's github page, and helped the open-source community patch the app [14], [7].

The top command server app, SSHDroid, is vulnerable to the local and Intranet cmdIFL attacks. This app works as an SSH server listening to the default port of 22, but it cannot prevent unauthorized connectors. An adversary does not need to fingerprint SSHDroid, because SSHDroid gives this information to any connector. SSHDroid only has two user name choices, the "user" in the normal case or the "root" on a rooted phone. It also uses the default "admin" password if the

TABLE VI
SERVERIFL SECURITY WEAKNESSES IN THE TOP 10 SERVER APPS.

Platform	App Id	App Name	Protocol	Port	Transmission Encryption	Authentication	Immune to File Upload CSRF	Effective Connection Alert	# of Installs*
Android	1	AirDroid	http	8888	× (setting)	✓ (user confirm)	✓	✓	10M - 50M
	2	WiFi File Transfer	http	1234	× (setting)	× (setting)	×	×	5M - 10M
	3	Xender	http	6789	×	✓ (four numbers)	✓	×	1M - 5M
	4	WiFi File Explorer	http	8000	×	× (setting)	◑	×	1M - 5M
	5	com.file.transfer	ftp	2121	×	×	✓	×	100K - 500K
iOS	6	Simple Transfer	http	80	×	× (setting)	✓	◑	1,504 Ratings
	7	Photo Transfer WiFi	http	8080	×	✓ (six bytes)	✓	×	865 Ratings
	8	WiFi Photo Transfer	http	15555	×	× (setting)	✓	×	661 Ratings
	9	USB & Wi-Fi Flash Drive	http	8080	×	×	×	×	462 Ratings
	10	Air Transfer	http	8080	×	× (setting)	✓	×	138 Ratings

* The app install numbers were counted on November 1, 2014. We use rating numbers to estimate the popularity of the iOS apps.

user does not change it in the settings. Hence, an adversary has enough pre-knowledge to exploit this app and steal its private files. More alarmingly, SSHDroid always tries to work as the "root". If it is exploited on a rooted phone, an adversary can execute root commands and steal the private files of all the installed apps.

D. ServerIFL Vulnerabilities

We test the top ten server apps from Google Play and the Apple App Store to evaluate the serverIFL vulnerabilities. Table VI summarizes the statistics of their security metrics. All of the tested apps have at least one security weakness that can be exploited to launch the serverIFL attacks.

Surprisingly, none of these apps provide encrypted transmission between file requesters (e.g., users' desktop browsers) and file servers (i.e., the tested apps). Eighty percent of the apps do not implement this important security guarantee at all. This can cause serious consequences in the wireless setting, which is assumed in these apps' user models. Two apps also provide this functionality which, however, is not enabled by default. We find that the apps' SSL encryption (when manually enabled in the setting) accepts only self-signed certificates, which causes security warnings in client-side browsers, thus hurting the user experience.

The authentication used in these apps is very weak. Seven of the ten tested apps do not enforce authentication, including the most popular iOS server app, Simple Transfer. An Intranet adversary can thus easily send unauthorized file extraction requests. The apps that conduct authentication still do not have guaranteed security due to the aforementioned lack of encrypted transmission. An adversary can sniff wireless traffic to obtain the secret information used for authentication or the cookies used for post-authentication transmission. The secret information used for authentication is generally not strong, such as the four-number verification code used in app #3 and the six-character password in app #7. Brute-force attacks are therefore practical. For example, to crack Xender's authentication, an adversary only needs to try 10,000 times at most.

Remote serverIFL attacks are also possible. We propose an improved file upload CSRF (cross-site request forgery) attack [8] for this purpose. An adversary uploads an HTML or PDF file with malicious JavaScript codes through the file upload CSRF. After the victim opens the uploaded file in his desktop browser, the embedded JavaScript runs in the same domain as other target files and can steal arbitrary file content. We only describe the test results here:

- Apps #2 and #9 suffer from file upload CSRF, making remote serverIFL attacks possible.
- Apps #4, #5, and #8 do not support file upload functionalities, making them immune to file upload CSRF. However, the paid version of app #4 supports file uploading.
- Apps #6 and #7 allow only uploading photo files, making them non-vulnerable.
- Apps #3 and #10 do not support viewing uploaded files. Therefore, the attack vector (i.e., the embedded JavaScript) cannot be executed in the victim's desktop browser.
- One app, AirDroid, embeds a secret token into each GET/POST request URL. As CSRF cannot obtain this token, the app is safe.

Launching stealthy serverIFL attacks usually requires that victim apps do not have an effective mechanism to detect illegal connections. Our evaluation reveals that only two apps have such detection capabilities. AirDroid alerts users to confirm or reject each new incoming connection and breaks the last connection. This mechanism is effective in preventing stealthy connections, because it is hard for a local attack app to disable or envelop AirDroid alerts. Simple Transfer displays a "connected" UI when it accepts its first connection. However, it does not further implement multiple-connection detections or warnings. This weakness allows an adversary to stealthily connect to a victim app after the victim has established its initial connection with a legal user browser.

To launch effective serverIFL attacks, an attacker could fingerprint common server apps in advance. As shown in Table VI, the protocols used in the tested apps are quite indistinguishable (basically HTTP). However, the opened ports have sufficient variability for identifying the apps. Moreover, for the apps with the same port numbers, an adversary can leverage different HTTP responses to distinguish them. Once the adversary has constructed a database of fingerprints, it can launch targeted serverIFL attacks on the apps.

V. ANDROID VS IOS

Our evaluation reveals four major differences between Android and iOS in terms of the impact of the IFL attacks. We

discuss their implications below.

Implication 1: *The common practice in iOS apps to open (untrusted) files in their own app domain could lead to more pervasive and powerful* `sopIFL` *attacks on iOS than Android.*

We compare the file-opening behavior in the iOS and Android versions of representative apps in different categories. We choose two file types, HTML and PDF, for their ability to carry attack vectors. We find that most of the tested iOS apps open the HTML files by themselves. In contrast, the corresponding Android versions choose to let dedicated apps (e.g., browsers) handle the HTML files. The PDF files have similar results. Opening untrusted files within the app's own domain is thus a common practice in iOS apps, whereas Android apps generally ask dedicated apps to open files. However, Google Drive and WeChat for iOS also require explicit user actions to open HTML files outside the app. But similar to other iOS apps, they open PDF files internally.

This common practice produces more attack surfaces for iOS apps than their Android versions. Asking dedicated apps to handle untrusted files is a more secure design, because potential attack vectors are then kept away from the user's private files. The tested Android apps generally use this practice, which makes them immune to `sopIFL` attacks. Hence, `sopIFL` attacks on Android are nearly local attacks that force files opened in *exposed* browsing interfaces, which only affects browser apps and careless apps. In contrast, iOS cases are more pervasive and span multiple app categories. They are also more powerful and can be local, web, or remote attacks, as they do not necessarily require locally exposed components.

There are many possible reasons for iOS's this design. We believe that the lack of flexible data sharing on iOS is an important reason responsible for the apps to open files by themselves. Indeed, the iOS data sharing involves a non-lightweight process of "exporting and importing," possibly due to the lack of public SD cards and a content URI mechanism, both of which are supported on Android.

Implication 2: *The randomized app data directory on iOS makes it difficult to conduct the* `aimIFL-1` *attacks on iOS.*

The `aimIFL-1` attacks usually require the knowledge of a full file path. However, iOS assigns a random directory for each app's data zone, which makes it difficult for a remote attacker to construct the full path of a target file. Moreover, this iOS randomness is performed at every installation. Therefore, the directory of an app reinstalled on the same phone will be different after each new installation. An example of a randomized app directory is `3570E343-2A5A-484E-BC86-7B3CC611D634`, with the unified path prefix `/private/var/mobile/Applications/` (on iOS 7) and `/private/var/mobile/Containers/Bundle/Application/` (on iOS 8).

In comparison, Android names app data directories according to the app package name. An adversary can easily construct the app directory using the pattern `/data/data/packagename/`. As apps generally do not use their own randomness within this directory, it is straightforward to obtain the full file path. We only find one exception: Firefox uses a random path strategy in its Android app design. An example of its full file path is `/data/data/org.mozilla.firefox/files/mozilla/62x7scuo.default/cookies.sqlite` with the randomized directory underlined.

As randomizing the app directory is useful for thwarting the `aimIFL-1` attacks on iOS, we recommend the Android developers to use this practice in their own app design.

Implication 3: *Apple's strict app review prevents iOS apps from executing bash commands. An adversary therefore cannot find targets to launch the* `cmdIFL` *attacks on iOS.*

As stated in [45], Apple has strict regulations for reviewing iOS apps submitted to the App Store. Apple's app review guidelines [2] briefly describe many scenarios that can lead to an app rejection. Among them, *rule 2.8* states:

> *Apps that install or launch other executable code will be rejected.*

This rule implies that running interpreted codes (e.g., bash scripts) is forbidden by Apple. We thus cannot locate any iOS apps that contain command interpreters, which is a necessary condition for launching the `cmdIFL` attacks. Although a few iOS apps (e.g., ipash ME) claim that they provide command execution for iOS, they actually only mimic the output and do not run the native commands. They therefore receive customer reviews saying "It's a fake command line."

To sum up, this iOS restriction makes it nearly impossible to launch the `cmdIFL` attacks on iOS, because there are no suitable app targets in the App Store.

Implication 4: *iOS generally does not allow background server behavior, which reduces the chance of the* `serverIFL` *attacks on iOS.*

The success of launching the `serverIFL` attacks depends on whether the adversary can attack victim apps when the phone screen is off or locked. If victim apps do not support background servers, then the attack timing window is shortened, thus reducing the chance of a successful `serverIFL` attack. The evaluation in Section IV-D indicates that iOS server apps usually only work in the foreground. Of the top five iOS server apps, only Air Transfer can be attacked when the screen is off. In contrast, all of the top five Android apps support background server behavior and are thus exploitable in the same phone setting.

We find that it is not easy for iOS developers to implement background server behavior. They require advanced tricks [10], [15] and have to worry about violating Apple app review policies. Thus, developing an app server that can run in the background is uncommon on iOS.

VI. Mitigation Methods

Application-specific defenses are required to mitigate existing IFL risks. Developers can refer to Section IV for avoiding the same flaws shown in the existing vulnerable apps. System flaws in Android and iOS, such as the SOP flaw mentioned in Section III-B, should be also timely fixed. Four implications in Section V will be useful to improve both app and system security at different levels. For example, it is prudent for iOS

apps not to open untrusted files in their own app domain and instead to ask dedicated apps to handle them.

We now offer two more suggestions, *NoJS* and *AuthAccess*, to further mitigate IFL attacks.

- NoJS: disabling JavaScript execution in local schemes to safeguard against the `sopIFL` and `aimIFL` attacks. The `serverIFL` attacks based on file upload CSRF can be similarly stopped by opening uploaded files as plain texts.
- AuthAccess: restricting commands and network requests to access apps' private file zone. Each access should be explicitly authorized by users. By doing so, the `cmdIFL` and `serverIFL` attacks can be mitigated.

Beyond vendors' own ad hoc fixes, a central mitigation solution is desirable. A possible way is to extend the existing SEAndroid MAC system [41] by leveraging the fine-grained context information to differentiate IFL attack requests from normal requests. Prior works [28], [26], [20], [21] have shown how to collect and enforce process-related context for tackling local permission leak attacks. The local IFL attacks can be handled in a similar way. However, it is challenging for context-based enforcement to mitigate the remote IFL attacks because remote entities are usually not under defender's control and thus their context cannot be easily obtained. To address this problem, recently proposed user-driven and content-based access control [40], [37] may be useful. We will investigate how to leverage them to develop an enhanced context-based MAC system for the IFL attack mitigation in our future work.

VII. Related Work

File leaks in mobile apps. Compared with the IFLs studied in this paper, direct file leaks are a more straightforward type of file leak. Many of these leaks were caused by the setting an insecure (e.g., world-readable) permission for its private files in the apps. For example, Opera [12] and Lookout [16] have made this error. On the other hand, the victim app writes sensitive files to public storage (e.g., SD cards and system debug logs). Outlook [13] and Evernote [6] put their users at risk in this way. The recent SEAndroid MAC system defends against these direct leaks, whereas our IFL is still an unsolved threat. Moreover, these direct cases are just local leaks on Android, whereas we propose multiple forms of remote file leaks across both Android and iOS.

Some IFL attack instances have been studied before, but they focus only on their specific problems. For example, Zhou et al. [51] study an attack with exposed Android content providers as confused deputies. By issuing unauthorized database queries to these components, an adversary can steal victim apps' database files. This attack is one kind of local IFL attacks and only exists on Android. Another example is our previous FileCross attacks [47], which belong to the `sopIFL` attacks discussed in Section III-B. However, it [47] only shows local `sopIFL` attacks on Android, whereas we demonstrate that `sopIFL` issues also exist in a number of iOS apps and can be remotely exploited.

It is worth noting that a blog post [11] reported two local `sopIFL` issues in Dropbox and Google Drive iOS apps over two years ago, but did not mention how to deliver the attack vectors (as we do in Section IV-A). This blog post used the old iOS systems before the recent iOS 7 and 8 to test the problem and did not show that this issue is widespread in the current iOS ecosystem. We are also the first to identify its fundamental cause: the legacy web SOP does not adequately cover the local schemes. Compared with all of these isolated IFL studies, we are the first to systematically study both local and remote IFL attacks across Android and iOS.

Confused deputy problems on mobile. The IFL attack is a class of the general confused deputy problem [31]. A number of previous works have studied the permission-related confused deputy problem on Android, called permission leak or privilege escalation [25]. They have proposed detection systems based on control- and data-flow analysis, including ComDroid [23], Woodpecker [30], CHEX [35], DroidChecker [22], Epicc [39], and SEFA [48]. Some Android app analysis frameworks, such as FlowDroid [18] and Amandroid [46], can be extended to detect this problem. However, it is difficult for these static tools to analyze the IFL vulnerabilities, because most of the IFL attacks do not have explicit vulnerable code patterns. Using dynamic analysis tools (e.g., [27], [49], [42], [19]) to construct automatic IFL detectors is therefore desirable. We leave this for our future work. Furthermore, none of the aforementioned studies identify confused deputy problems on iOS. But as shown in our IFL attacks, it is equally, if not more, important to develop detection tools for iOS.

Mobile browser security. Our `sopIFL` and `aimIFL` attacks are related to mobile browser security. Related works have studied the threats to Android WebView [36], [24], [38], [47], the security risks in HTML5-based mobile libraries [29] and apps [33], and unauthorized origin crossing in several popular Android and iOS apps [44]. Differently, our focus is file leaks via vulnerable browser components. Only the aforementioned work [47] shares the same goal as our study. In addition, a prior work [33] injects unauthorized JavaScripts into HTML5-based apps, and their technique is similar to our `aimIFL` attacks. However, as their goal is to compromise the victim website's online credentials (instead of our goal of stealing local files), they do not need to overcome our challenge of launching local file-stolen attacks from a web origin due to the SOP restriction. Moreover, our `aimIFL` attacks apply to all mobile apps that contain browser components, rather than just the HTML5-based apps shown in [33].

VIII. Conclusion

In this paper, we systematically studied indirect file leaks (IFLs) in mobile applications. In particular, we devised four new IFL attacks that exploit browser interfaces, command interpreters, and embedded app servers to leak private files from popular apps. Unlike the previous attacks that work only on Android, we demonstrated that these IFLs (three of them) can affect both Android and iOS. The vulnerable apps include Evernote, QQ, and Mail.Ru on iOS, and Baidu Browser, 360

Mobile Safe, and Terminal Emulator on Android. Moreover, we showed that our four IFL attacks can be launched remotely, without implanting malicious apps in victim's smartphones. This remote attack capability significantly increases the impact of the IFL attacks. Finally, we analyzed the differences between Android and iOS in terms of the IFL attacks' impacts and proposed several methods to mitigate the attacks.

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

Additional materials. We will provide supplementary materials, such as detailed vulnerability reports, at this link (https://daoyuan14.github.io/pp/most15.html).

REFERENCES

[1] According to Apple, people have all but stopped upgrading to iOS 8. http://9to5mac.com/2014/10/06/ios8-market-share-stagnated/.

[2] App store review guidelines. https://developer.apple.com/app-store/review/guidelines/.

[3] Apps have overtaken the web in popularity according to the latest statistics. http://www.dailymail.co.uk/sciencetech/article-2119332/Apps-overtaken-Web-popularity-according-latest-statistics-actually-theres-probably-app-tell-that.html.

[4] Apps more popular than the mobile web, data shows. http://www.theguardian.com/technology/appsblog/2014/apr/02/apps-more-popular-than-the-mobile-web-data-shows.

[5] CVE-2014-0521. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0521.

[6] Evernote patches vulnerability in Android app. http://blog.trendmicro.com/trendlabs-security-intelligence/evernote-patches-vulnerability-in-android-app/.

[7] Fixing issue #374 in android-terminal-emulator. https://github.com/jackpal/Android-Terminal-Emulator/commit/51129616.

[8] How to upload arbitrary file contents cross-domain. http://blog.kotowicz.net/2011/04/how-to-upload-arbitrary-file-contents.html.

[9] iOS apps created with PhoneGap. http://phonegap.com/app/ios/.

[10] iOS background application network access. http://stackoverflow.com/questions/9613357/ios-background-application-network-access.

[11] Old habits die hard: Cross-zone scripting in dropbox & google drive mobile apps. http://blog.watchfire.com/wfblog/2012/10/old-habits-die-hard.html.

[12] Opera mobile for Android insecure file permissions cache poisoning vulnerability. http://www.securityfocus.com/bid/49702/discuss.

[13] Outlook Android app stores emails in plain text on mobile. http://securityaffairs.co/wordpress/25103/digital-id/outlook-app-leaks-encryption.html.

[14] Pull request #375 in android-terminal-emulator. https://github.com/jackpal/Android-Terminal-Emulator/pull/375.

[15] Technical note TN2277: Networking and multitasking. https://developer.apple.com/library/ios/technotes/tn2277/.

[16] Vulnerability id: Look-11-001. https://blog.lookout.com/look-11-001/.

[17] S. Acker, N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proc. ACM AsiaCCS*, 2012.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM PLDI*, 2014.

[19] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *Proc. Usenix Security*, 2014.

[20] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on Android. In *Proc. ISOC NDSS*, 2012.

[21] S. Bugiel, S. Heuser, and A. Sadeghi. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. In *Proc. Usenix Security*, 2013.

[22] P. Chan, L. Hui, and S. Yiu. DroidChecker: Analyzing Android applications for capability leak. In *Proc. ACM WiSec*, 2012.

[23] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proc. ACM MobiSys*, 2011.

[24] E. Chin and D. Wagner. Bifocals: Analyzing webview vulnerabilities in Android applications. In *Proc. Springer WISA*, 2013.

[25] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proc. Springer ISC*, 2010.

[26] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proc. USENIX Security*, 2011.

[27] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. Usenix OSDI*, 2010.

[28] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *Proc. Usenix Security*, 2011.

[29] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Proc. ISOC NDSS*, 2014.

[30] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proc. ISOC NDSS*, 2012.

[31] N. Hardy. The confused deputy: (or why capabilities might have been invented). In *ACM SIGPOS Operating Systems Review*, 1988.

[32] Y. Jang, C. Song, S. Chung, T. Wang, and W. Lee. A11y attacks: Exploiting accessibility in operating systems. In *Proc. ACM CCS*, 2014.

[33] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Proc. ACM CCS*, 2014.

[34] C. Lin, H. Li, X. Zhou, and X. Wang. Screenmilker: How to milk your Android screen for secrets. In *Proc. ISOC NDSS*, 2014.

[35] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proc. ACM CCS*, 2012.

[36] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the Android system. In *Proc. ACM ACSAC*, 2011.

[37] A. Moshchuk, H. Wang, and Y. Liu. Content-based isolation: Rethinking isolation policy design on client systems. In *Proc. ACM CCS*, 2013.

[38] A. Nadkarni, V. Tendulkar, and W. Enck. Nativewrap: Ad hoc smartphone application creation for end users. In *Proc. ACM WiSec*, 2014.

[39] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proc. Usenix Security*, 2013.

[40] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[41] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing flexible MAC to Android. In *Proc. ISOC NDSS*, 2013.

[42] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proc. ISOC NDSS*, 2014.

[43] T. Terada. Attacking Android browsers via intent scheme urls. http://www.mbsd.jp/Whitepaper/IntentScheme.pdf, 2014.

[44] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proc. ACM CCS*, 2013.

[45] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on iOS: When benign apps become evil. In *Proc. Usenix Security*, 2013.

[46] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *Proc. ACM CCS*, 2014.

[47] D. Wu and R. Chang. Analyzing Android Browser Apps for file:// Vulnerabilities. In *Proc. Springer Information Security Conference (ISC)*, 2014.

[48] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on Android security. In *Proc. ACM CCS*, 2013.

[49] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, 2012.

[50] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. IEEE Symposium on Security and Privacy*, 2012.

[51] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in Android applications. In *Proc. ISOC NDSS*, 2013.