Privacy Leakage and Protection of InputConnection Interface in Android

Chi-Yu Li*, Hsin-Yi Wang†, Wei-Ching Wang‡, and Chun-Ying Huang*
* College of Computer Science, Department of Computer Science,
National Chiao Tung University & National Yang Ming Chiao Tung University
† Verizon Media, Inc. ‡ Trend Micro, Inc.

***Abstract*—**
Leakage of user credentials has been a conventional security threat for mobile users. In this work, we discover a new leakage threat caused by a vulnerability of the input method framework (IMF) on Android. The vulnerability lies in an IMF interface, called InputConnection, which is dynamically built to deliver user inputs from an active input method (e.g., software keyboard) to WebView, which is an essential Android component rendering web pages. It allows the IMF interface of a WebView component to be hijacked by the app or the third-party library that embeds the WebView. Such hijacking can be exploited to steal user inputs on the web pages loaded by the WebView. It can also eavesdrop on input fields of all the web pages loaded by WebView without user awareness; the attack is self-contained and does not require any external dependency. It does not interrupt, delay, or change normal operations. More threateningly, this attack is easy to launch and works for most Android versions (from 4.4 to 11.0). We conduct a field study including more than 1500 tests on our developed IWH attack app. The result shows that the app can successfully steal user inputs in all the tests and identify the input strings with 98.0% accuracy. We further devise two solutions, a web-based virtual keyboard and an IMF hijacking guardian, for mobile web services and the Android platform, respectively. We finally prototype them on a web server and on an Android framework, respectively, to confirm their effectiveness.

***Index Terms*—Android, Information Leakage, Input Method Framework, Mobile Privacy, WebView**

I. Introduction

Mobile devices have become the major portal for users to access the Internet. Statistics [1] show that the Internet usage of mobile and tablet devices has exceeded that of desktops since 2016. Mobile-friendly web design has also become one important ranking factor considered by search engines [2]. However, due to limited display sizes and constrained input interfaces, it is difficult for mobile users to identify possible threats when surfing the Internet. It thus has led mobile devices to become a new playground for attackers.

Information leakage is one critical issue for Internet surfing on mobile platforms. Attackers can launch phishing [3]–[5] and pharming [6] attacks to steal sensitive information from users. Specifically, phishing attacks trap users to provide their credentials using malicious links and forged websites. The pharming ones steal user credentials by redirecting users to malicious sites based on the hijacking of name resolution services. A number of research works have focused on solving these attacks [7]–[15]. However, statistics [16], [17] show that the number of phishing attacks continues to grow in recent years. The total number of phishing attacks in 2016 was more than 1.2 million, with a 65% increase over 2015. It shows that the financial gains from the information leakage attacks are still attractive.

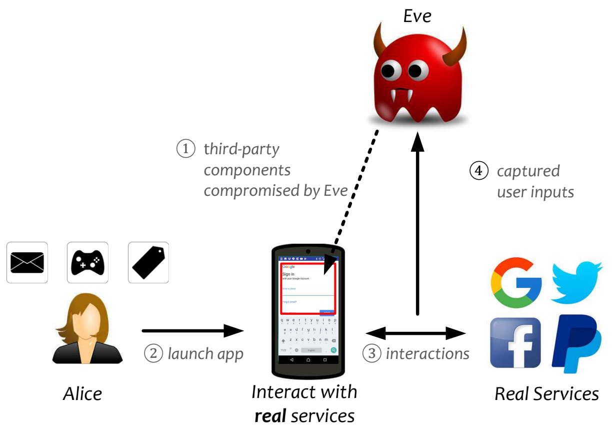
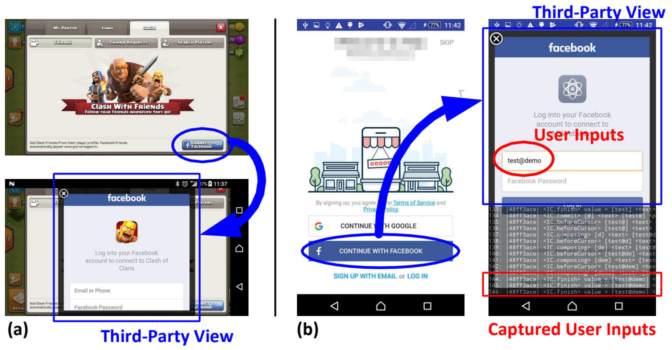

Eve
① third-party components compromised by Eve
④ captured user inputs
② launch app
③ interactions
Alice
Interact with **real** services
Real Services

Fig. 1. An invisible WebView hijacking attack against the common account binding scenario.

In this paper, we discover a vulnerability that can leak user inputs from arbitrary rendered third-party web content via the Android input method framework (IMF). The vulnerability lies in an IMF interface, called InputConnection, which is dynamically built to deliver user inputs from an input method (e.g., software keyboard) to an Android view. It allows the InputConnection interface of a WebView component, which is used to render web pages on Android, to be hijacked by the app or the third-party library that embeds the WebView. That is, all the inputs delivered from an input method to a WebView component can be leaked. However, it shall be prohibited and can be attributed to a design defect of Android IMF. Note that through Android APIs, an app is not allowed to fetch user inputs on the web pages rendered by its WebView component.

We exploit this vulnerability to devise an attack, *invisible WebView hijacking (IWH)*, which can eavesdrop on all the web pages with input fields without user awareness. It can be launched by malicious apps or third-party components (e.g., advertisement libraries and SDKs), which are possibly used by benign apps. Literature has shown that the use of third-party components is common in application development [18]–

Corresponding author: Chun-Ying Huang (email: chuang@cs.nctu.edu.tw; chuang@nycu.edu.tw).



```
133: |48ff3ace| <IC.finish> value = [test] <text>
134: |48ff3ace| <IC.commit> [@] <text> [test@] <p
135: |48ff3ace| <IC.beforeCursor> [test@] <text>
136: |48ff3ace| <IC.beforeCursor> [test@] <text>
137: |48ff3ace| <IC.composing> [d] <text> [test@d
138: |48ff3ace| <IC.beforeCursor> [test@d] <text>
139: |48ff3ace| <IC.composing> [de] <text> [test@
140: |48ff3ace| <IC.beforeCursor> [test@de] <text
141: |48ff3ace| <IC.composing> [dem] <text> [test
142: |48ff3ace| <IC.beforeCursor> [test@dem] <tex
143: |48ff3ace| <IC.composing> [demo] <text> [tes
144: |48ff3ace| <IC.finish> value = [test@demo] <
145: |48ff3ace| <IC.finish> value = [test@demo] <
146: |48ff3ace| <IC.finish> value = [test@demo] <
```

Fig. 2. Two gaming apps request user authentication through the user's Facebook account based on the OAuth framework using the WebView. (a) A benign app. (b) A malicious app that wiretaps user credentials.

[21]. Moreover, it is not rare for third-party components to be compromised [22], [23]. They may trap users to login pages (e.g., Facebook OAuth) and steal user credentials without the awareness of users, benign apps, or anti-malware scanners.

A simple scenario of the IWH attack is shown in Figure 1. The adversary, *Eve*, can compromise several third-party libraries with the IWH attack. Once any benign apps use those compromised libraries, the users who use the apps may suffer from the leakage of user credentials. Consider *Alice* uses an app developed based on an IWH-compromised library and launches its embedded browser to perform web-based account binding from an Internet service provider (e.g., Google, Facebook, and Twitter). While *Alice* goes through the account binding process with the input of her user credential, *Eve* can stealthily capture the user credential without getting *Alice*'s awareness. Note that it is common for modern mobile apps to perform the account binding; according to Facebook [24], more than 25% of app users choose to connect with Facebook to improve user experience. In addition to the OAuth scenarios, other possible scenarios that require a third-party component to render arbitrary remote content include customized WebViews (for example, from ad-block SDKs), advertisement SDKs, and Google Play instant applications.

Take the gaming app as an example. Figure 2 shows two gaming apps, which request user authentication through the user's Facebook account based on the OAuth framework[1]. The left one is a benign app, and the right one wiretapping user credential is malicious. The latter can be a malicious app or a benign app that uses a malicious third-party OAuth SDK. It can leak the user's Facebook credential to the adversary. Note that the view of captured user inputs is for demonstration only but can be hidden during attacks.

In contrast to traditional keylogging threats, IWH does not require any additional permission, setup, or root privilege. Unlike conventional phishing and pharming attacks, it does not have to mimic an app, forge a website, perform redirections, or hijack name resolution services. Therefore, it cannot be detected by existing detection mechanisms [7]–[11], [25]–[27]. All the solutions proposed for these existing attacks cannot defend against IWH. Although IWH requires users to install a malicious app as conventional phishing attacks [3], [4], including app spoofing and GUI confusion, it is much more threatening from two security aspects. First, the effort of launching IWH is much less than those phishing attacks since the latter attacks need to customize a set of views or GUIs for each original app (e.g., Facebook). Our attack does not need such customization but generally works for the web pages loaded by WebView. Second, those conventional attacks may easily cause user awareness and then fail from the user's precautions, since the spoofed pages or the mimicked GUIs may not show correct user information or valid response after a user credential is submitted (e.g., user-specific information after a successful login), or some GUI confusion actions (e.g., app switch) may not work smoothly. Instead of intervening in the user interaction, our IWH attack is more stealthy by passively eavesdropping on the input channel.

Our prototype shows that the IWH attack can be used to successfully wiretap user inputs without being noticed. It is stealthy that under attacks, users are still allowed to access services just as normal — valid hostnames, valid web certificates, valid content, and no alarms! Our field study shows that the attack is easy to launch and works for most Android versions (from 4.4 to 11.0) on various smartphones with seven phone brands and 22 device models. From more than 1500 tests, it is observed that our developed attack app can successfully steal user inputs in all the tests and identify the input strings with 98.0% accuracy.

We devise two solutions to address the vulnerability: a web-based virtual keyboard and an IMF hijacking guardian for mobile web services and the Android platform. The virtual keyboard prevents IMF from being used for the need of user input. The hijacking guardian sets up a security association between two ends for the delivery of user inputs. It encrypts the user inputs in transit so that they cannot be leaked even if the InputConnection interface is hijacked. In our prototype, it is observed that the delay imposed by the guardian is negligible.

Our contribution is threefold. First, we discover a vulnerability[2], input channel hijacking, from the IMF framework and analyze its root causes. Second, we exploit the vulnerability to devise an IWH attack that enables malicious apps or third-party components to wiretap user inputs stealthily from WebView. Third, we propose solutions that can immediately mitigate the IWH attack from the perspectives of both mobile web services and the Android system.

The rest of this paper is organized as follows. We survey related works in Section II and introduce the background of IMF and WebView in Section III. Section IV describes our threat model and methodology. Section V presents the vulnerability of input channel hijacking from the IMF and the IWH attack. We propose solutions and discuss issues in Sections VI and VII, respectively. Section VIII concludes the paper.

[1]There are two ways to implement OAuth in an Android application: web services and Android APIs. A malicious app can exploit the vulnerability by relying on the former with WebView.

[2]We follow the principle of responsible disclosure by reporting the discussed vulnerability to Android. The submission date of this paper was beyond Google's 90-full-day disclosure policy [28].

II. Related Work

As Android becomes the most popular mobile OS worldwide, a great number of research studies have examined its security issues from various aspects. They mainly focus on the vulnerabilities of system components (e.g., WebView [29]–[34], Binder[3] [35], keyboard apps [21], [36] and services [37]–[41], and hover technology [42]) and existing security mechanisms (e.g., permission/access control [43]–[46], password protection [47], [48], code injection avoidance [49], and privacy-preserving [50]). We here focus on security studies of the WebView component and the threats of privacy leakage on Android.

WebView Security. Several studies have addressed some vulnerabilities of WebView Security. Specifically, Shin et al. [29] discovered that WebView-based apps are vulnerable to the SSL stripping attack and then proposed a solution of visual security cues against it. Li et al. [51] revealed a new threat based on cross-app WebView infection. A malicious attacker can trap a user to click a crafted URL and invoke WebView embedded in a vulnerable app, and then use WebView's JavaScript interfaces to instruct the compromised app to perform some specific attacks. Another study [34] examined that attackers may compromise the web content shown in the WebView using the APIs which WebView inherits from general-purpose UI components. Most of the other works [30]–[33], [52]–[55] focus on the WebView's weakness that JavaScript codes in mobile web pages are allowed to gain custom interfaces exposed by apps and may thus be able to manipulate the device or steal sensitive data. Luo et al. [32] introduced several ways to exploit this weakness, and other studies [30], [31], [33], [55] seek to address it using access control-based mechanisms. Different from all these existing works, we focus on the vulnerability of the cooperation between WebView and the IMF framework.

Privacy Leakage on Android. The privacy leakage threats can be classified in terms of exploited vulnerabilities. Third-party keyboard applications may launch keylogger attacks [37], [39]–[41] to steal sensitive information from Android users. Several studies [41] seek to detect whether there are any risks of being keylogged by the keyboard applications, whereas another work [40] defends against the keylogger based on one-time passwords. Third-party applications may leak users' private data once they are authorized to access them [56]–[58]. Android users may still suffer from conventional phishing and pharming attacks, but several solutions [7]–[10], [12]–[15], [25], [26], [59], [60] have been proposed.

Studies [4], [36], [38], [61]–[63] explore the privacy leakage threats from the vulnerabilities of motion sensors, the Accessibility feature, clipboard, and Android GUI. Specifically, Cai et al. [38] can employ motion sensors to infer keystrokes based on the phenomenon that pressing different locations of the software keyboard causes dissimilar vibrations. Fratantonio et al. [61] present that it is possible to trap users to activate the Accessibility feature, which can be abused to implement a key logger. Fahl et al. [62] discover that applications without any permission can access the clipboard. Chen et al. [63] discuss possible leakage of sensitive information from an untrusted input method editor (IME). They further propose an app-transparent sandbox to confine IMEs to predefined securities. Diao et al. [36] attempt to harvest entries from the personalized user dictionary of IME. Bianchi et al. [4] study GUI confusion attacks that can replace or mimic the GUI of benign apps to steal sensitive information and then design an on-device defense that can securely inform users about the origin of the app with which they are interacting. Different from them, our privacy leakage attack exploits a new vulnerability lying in the IMF framework.

[3]Binder is a class for interprocess communication on Android.

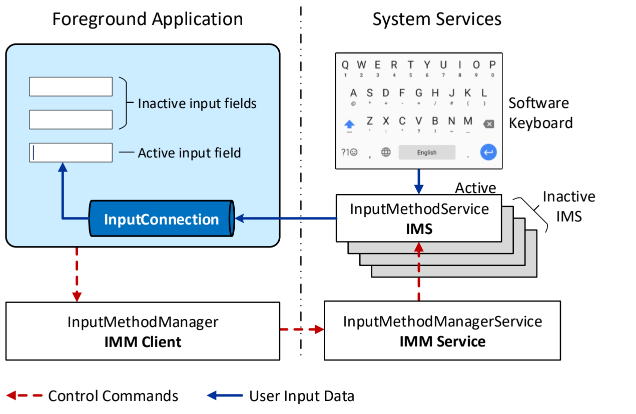


Fig. 3. The IMF architecture of a case that the foreground app is connected with an input method, a software keyboard.

UiRef [64], SUPOR [65], and UIPicker [66] were designed to automatically check whether an Android app requests sensitive data. They detect sensitive input fields by performing static analysis against source codes as well as layout resource files. These works are effective for inspecting native text input UI components. However, these static checkers are not able to handle possible data leakage from the input fields embedded in a WebView component.

III. Background

In this Section, we introduce two Android components: IMF and WebView. IMF enables user inputs on an Android view, which displays content and carries interactive UI components (e.g., buttons and text fields). WebView, a kind of view, is employed to display web content by an app.

A. *Input Method Framework*

IMF [67] provides arbitration of interactions between apps and input methods. It contains three primary parties: input methods, an input method manager (IMM), and apps. There are various input methods, e.g., software keyboards, hardware keyboards, and hand-writing recognizers. Each method serves as a service, called input method service (IMS). As shown in Figure 3, only one IMS can be active at a time, whereas the others are inactive. Users can activate any of them from the picker dialog of input methods. The foreground app dynamically creates an InputConnection (IC) interface to receive user inputs from the active IMS whenever any of its editor components (e.g., a text input field) gets focus. IMM is responsible for this dynamic binding. The binding is removed upon the cancelation of the focus. IMM consists of two entities: IMM service and IMM client. The former is a global system service that provides user input service to the foreground app, whereas the latter residing at each app deals with the request of user input.

We illustrate the IMF operation based on an example case that the user starts to focus on a text input field in the foreground app, and the default input method is a software keyboard. As shown in Figure 3, the app issues a user input request through the IMM client as soon as the text input field is clicked. A delivery path of control commands (e.g., hiding and showing the software keyboard), which is shown as dotted lines, is then set up between the app and the IMS through the IMM. In the meantime, the app creates an IC interface to receive user inputs. It then passes the interface access to the active IMS via the control path. The data path, which is represented by solid lines, is thus built to span the software keyboard, the IMS, the IC interface, and the active input field. The interface can then read text around the input cursor and commit the user's keyboard inputs to the active input field. It is removed whenever the current input focus is canceled. Note that a new IC instance is created when the focus is switched to another input field.

Current Security Defense. The IMF system provides three major security defenses [67] against the leakage of user inputs. First, it ensures that only the foreground app's IC interface is allowed to bind to the IMS since malicious apps in the background may do the binding to wiretap the IMS to steal user inputs. Second, it prevents the foreground app from retrieving all text edits and most key events, which are sent to its embedded WebView component. They may be issued to form user inputs, which can be user credentials, on third-party web pages loaded by the WebView, so they should not be taken by the app. Third, the active input method has to be selected manually by the user; otherwise, malicious apps may programmatically switch to a malicious input method and then intercept user inputs.

B. WebView

WebView is an Android component, which can be embedded in an app to display web pages. It can be employed to load the pages stored locally in the app or remote Internet pages. Remote page loading is mainly used for two purposes. First, the developers seek to provide users the convenience of browsing third-party whose URL links are shown in their apps. For example, the Facebook app relies on WebView to show the page of a URL link clicked by a user. Second, they provide web content that is hosted by themselves on the Internet since the content can be easily changed without any app update. WebView offers some browser-like functions (e.g., navigating forward/backward, zooming in/out, and searching text). An app can also enable JavaScript functions to interact with the web content. Specifically, they can interact with its Java objects, which can be initially embedded or be injected into the WebView by the app. Ideally, any user inputs on third-party web pages rendered by WebView should not be allowed to be acquired by the app or the software component which embeds the WebView. However, using WebView may encounter the following security issues.

Current Security Issues and Defense. There are two major security issues with using WebView. First, malicious apps can perform JavaScript code injection attacks to steal user inputs on WebView [30]–[33], [52]–[54]. They can inject JavaScript codes to enumerate all input fields in a loaded web page by scanning the HTML/DOM objects of `<input>` and then retrieve their corresponding values. However, this attack can be addressed by the same-origin policy. It limits the run-time scope of JavaScript codes; that is, the codes running on a web page are not allowed to access the other origins' pages. From the WebView interface, any JavaScript codes can be injected into the main frame page rendered by the WebView, but the injection is not allowed in the other sub-frame pages. Therefore, a web service can avoid this injection attack by reorganizing web page layouts and creating barriers to the access of sensitive input fields.

Second, a malicious web app can attack WebView by reading or tampering with content inside a WebView object when they are in the same security context. However, it has been addressed by a new security feature introduced by Android 8.0 [68]. An app's WebView object is run in a separate process different from its main process, so they are in different security contexts and the attack can be prevented.

IV. Threat Model and Methodology

Threat Model. We focus on the scenario where mobile users input their user credentials on the web pages rendered by WebView for account login or binding purposes on Android devices. The adversary's objective is to capture the user credentials stealthily without getting the users' awareness. The adversary cannot modify any default Android objects including WebView, nor install any screenlogger or keylogger malware on the devices; it does not have their root access either.

The adversary can distribute a malicious component or app with only the `INTERNET` permission to Android devices. The component is in the form of a third-party library or SDK which provides some useful functions or convenient development kits, e.g., map/browser/advertisement libraries and OAuth SDKs. The victim devices install either a benign app that uses the malicious component or the malicious app; the apps use WebView to render web pages for users to perform account login or binding services. The malicious entities can passively steal user credentials using the IWH attack, but do not interfere with the operations of normal services, so the victims are unaware of the attack.

Methodology. We conduct all the feasibility tests and solution evaluations with our own phones in our laboratory. Together with the field study in Section V-C, we validate the identified vulnerability and attack using 22 Android phone models with Android versions from 4.4 to 11.0 and various software keyboards (see details in Table II). For the attack evaluation, we get IRB approved to invite normal users to test

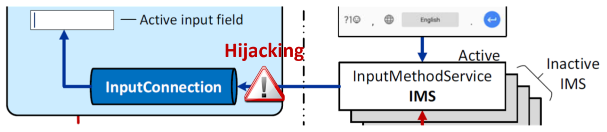


Fig. 4. Illustration of the input channel hijacking.

our developed malicious app and examine whether their input credentials can be successfully stolen or not. For each test, we notify the tester that his/her real credentials shall not be used. In case that the real credentials may be accidentally typed, we keep all the collected data safe without leaking them.

Note that we here focus on how to enable the attack, but how to distribute malicious components or apps, trap the victims to input user credentials, and abuse user credentials to cause damage is beyond the scope of this work.

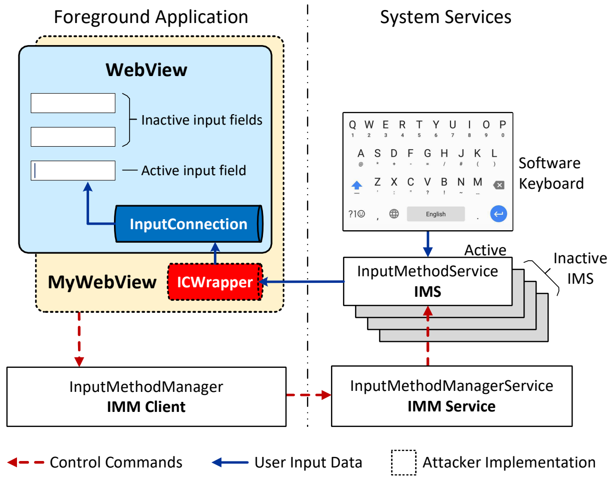


Fig. 5. An overview of the input channel hijacking.

V. Privacy Leakage from IMF

In this Section, we identify a vulnerability, input channel hijacking, from IMF. It can leak private user inputs from the web pages rendered by WebView. However, it shall be prohibited. In what follows, we present the vulnerability with an analysis of its root cause, devise an attack by exploiting it, and then conduct a field study for the attack.

A. Vulnerability: Input Channel Hijacking

We discover that the input channel between an editor and an active IMS can be easily hijacked. This hijacking can leak the user input on a web page rendered by WebView to the foreground app, as shown in Figure 4.

Seemingly, the input channel does not have any vulnerable points observed from its setup procedure and Android APIs. Upon an editor's focus, the IMM client gets an IC instance from the current view object by calling its `onCreateInputConnection` method. It then requests the IMM service to bind the instance to the active IMS (e.g., software keyboard). Afterward, the user input can start to be delivered from the IMS to the editor through the input channel, which contains the IC instance. In particular, no Android APIs are provided for the foreground app to access the IC or interact with the IMM client/service.

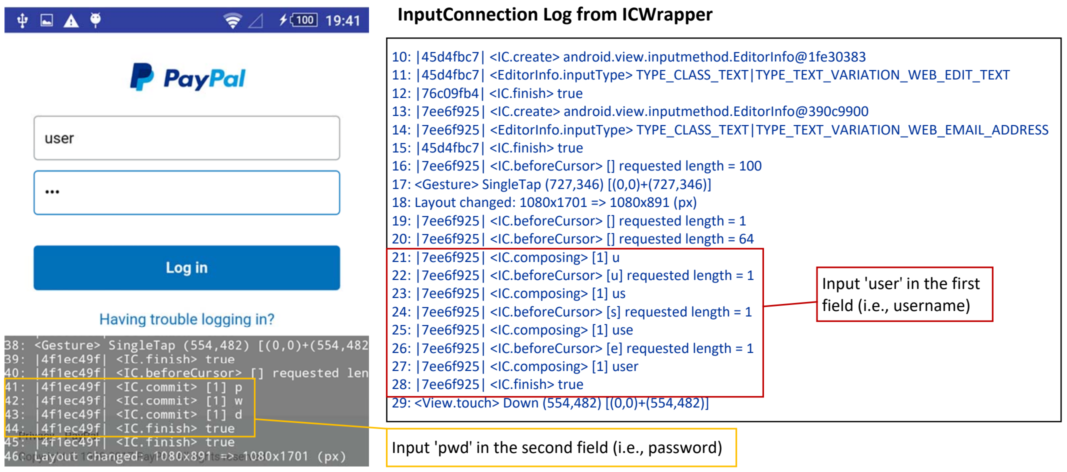
However, there exists a vulnerability that a foreground app can return its customized IC instance to the IMM client and let it bind to the active IMS since the WebView's `onCreateInputConnection` method can be overridden. The method is provided by the View class (i.e., `android.view.View`), and any object inheriting the View can override it to enable the function of user input. For example, the WebView class implements this method so that the user is allowed to type words inside WebView objects. To exploit this vulnerability, a malicious app can create a new WebView object which overrides the `onCreateInputConnection` method to give a customized IC instance to the IMM client. As a result, all the user inputs received by the IMS will be sent to the malicious app's customized IC.

To maintain normal operations, there are two requirements for the customized IC instance. First, it shall deliver user inputs to the editor as normal by forwarding them to the WebView's inherent IC instance, which connects to the editor. Second, it shall support the methods (more than 20) that the IMS can call. The methods include reading text around the cursor, committing text, etc. Thanks to the IMF support, there are two classes available for the customized IC: BaseInputConnection and InputConnectionWrapper. The former is the base case of the IC interface, whereas the latter is a wrapper class that can proxy function calls to an IC instance. The customized instance can be easily carried out based on either of them. In sum, the customized IC instance works as the middleware to proxy calls of the methods between the inherent IC instance and the IMS.

Validation. We show that the input channel of a WebView object can be hijacked by the foreground app, which embeds the WebView. We create an object, `MyWebView`, from the WebView class in our test app. There are two steps to enable the hijack. First, we implement the middleware, ICWrapper, by inheriting the InputConnectionWrapper class. Second, we override the `onCreateInputConnection` method of the WebView class. Figure 5 shows the relationship between different objects when the input channel is hijacked. When the IMM client requests an IC instance from the `MyWebView`, an ICWrapper instance is created and returned. Meanwhile, an IC instance of the WebView is also created, and then the ICWrapper forwards calls to it. Afterward, the ICWrapper can keep capturing the user input until the editor's focus is canceled. Upon the cancelation of the focus, both the IC and ICWrapper instances are dismissed.

We then use the `MyWebView` app to load the PayPal login page and validate that it can capture the text typed by the user on the page, as shown in Figure 6. The left figure is the app snapshot on a smartphone, whereas the right one is the log that we generate from the ICWrapper. Some log entries shown at the bottom of the snapshot are the latter part of the log. The numbers shown at the beginning of the log entries indicate their sequence.

Note that we generate the log from each function call mediated by our ICWrapper. Table I shows the tag we mark


19:41
PayPal
user
•••
Log in
Having trouble logging in?
InputConnection Log from ICWrapper
Input 'user' in the first field (i.e., username)
Input 'pwd' in the second field (i.e., password)

```
38: <Gesture> SingleTap (554,482) [(0,0)+(554,482
39: |4f1ec49f| <IC.finish> true
40: |4f1ec49f| <IC.beforeCursor> [] requested len
41: |4f1ec49f| <IC.commit> [1] p
42: |4f1ec49f| <IC.commit> [1] w
43: |4f1ec49f| <IC.commit> [1] d
44: |4f1ec49f| <IC.finish> true
45: |4f1ec49f| <IC.finish> true
46: Layout changed: 1080x891 => 1080x1701 (px)
```

```
10: |45d4fbc7| <IC.create> android.view.inputmethod.EditorInfo@1fe30383
11: |45d4fbc7| <EditorInfo.inputType> TYPE_CLASS_TEXT|TYPE_TEXT_VARIATION_WEB_EDIT_TEXT
12: |76c09fb4| <IC.finish> true
13: |7ee6f925| <IC.create> android.view.inputmethod.EditorInfo@390c9900
14: |7ee6f925| <EditorInfo.inputType> TYPE_CLASS_TEXT|TYPE_TEXT_VARIATION_WEB_EMAIL_ADDRESS
15: |45d4fbc7| <IC.finish> true
16: |7ee6f925| <IC.beforeCursor> [] requested length = 100
17: <Gesture> SingleTap (727,346) [(0,0)+(727,346)]
18: Layout changed: 1080x1701 => 1080x891 (px)
19: |7ee6f925| <IC.beforeCursor> [] requested length = 1
20: |7ee6f925| <IC.beforeCursor> [] requested length = 64
21: |7ee6f925| <IC.composing> [1] u
22: |7ee6f925| <IC.beforeCursor> [u] requested length = 1
23: |7ee6f925| <IC.composing> [1] us
24: |7ee6f925| <IC.beforeCursor> [s] requested length = 1
25: |7ee6f925| <IC.composing> [1] use
26: |7ee6f925| <IC.beforeCursor> [e] requested length = 1
27: |7ee6f925| <IC.composing> [1] user
28: |7ee6f925| <IC.finish> true
29: <View.touch> Down (554,482) [(0,0)+(554,482)]
```

Fig. 6. Our `MyWebView` app loading the PayPal login page can wiretap the user input. The left figure is the app snapshot on a smartphone, whereas the right is the log outputted from the ICWrapper instance.

in each function and its description. In each function, we can collect the characters the user is typing and the status of the current editor (or input field). When the user types a character, the IMS calls the function "setComposingText" to replace the currently composing text with an updated one or the function "commitText" to commit the character. In the former function, marking composition enables the IMS to provide a function of word suggestions, where the user can select a suggested word to finish typing after only a few characters are typed. The newly composing text can be either the old text plus the character typed by the user or a suggested word. The latter function is used for the cases that word suggestions are not applicable, such as password fields (e.g., the password text on the left side of Figure 6). Moreover, some other functions of the IC instance are provided for the IMS to monitor the editor's status. For example, we can obtain the finish of the editor, current text in the editor, and the text before the current cursor position from the functions "finishComposingText," "getExtractedText," and "getTextBeforeCursor," respectively.

As shown on the right side of Figure 6, the user input "user" in the username field can be captured in log entry 27. The characters following the tag `IC.composing` are those typed by the user and sent from the IMS to the `MyWebView` via the ICWrapper. The content following the tag `IC.beforeCursor` includes the characters before the current cursor and the requested length. It is requested by the IMS and then sent from the IC instance to it. When the composing is done, the IMS sends a finish message to the IC. Different from the username field, the IMS uses the commit function instead of composing to send input text to the IC for the password field. As shown in Figure 6, we are able to steal both username and password strings from the generated log.

Root Cause. This vulnerability is rooted in two design issues and can be attributed to a design defect of IMF. The first is *insecure delivery of user input data*. The security shall be either to guarantee that no malicious entities can hijack or wiretap the delivery path or to provide an end-to-end security association against malicious entities which may sit in the middle. The discovered vulnerability shows that the current IMF design does not have either of them. It has no security association between two separate entities that communicate in the IMF framework. This kind of communication can usually happen in a system and should be carefully examined in terms of data leakage concerns. Specifically, those two ends are the active IMS and the WebView's inherent IC instance, which connects to the focused editor. Since the WebView's `onCreateInputConnection` method can be overridden, an intermediate entity, a customized IC instance (i.e., the ICWrapper in Figure 5), can be built to sit between them by the foreground app.

The second issue is that *two functions are coupled by a method so that no flexibility exists for disabling them individually*. Specifically, they are the IC interface initialization and the IMS configuration, which are coupled by the `onCreateInputConnection` method. An intuition solution is to prevent WebView's `onCreateInputConnection` method from being overridden, but it can lead to two critical issues and be thus hardly considered. First, it would disable the IMS configuration, which allows an app to restrict the IMS to accept only numbers or letters as user inputs for each IC instance. Second, some apps have been developed based on the override, so the change requires them to be updated.

B. Invisible WebView Hijacking Attack

We devise an invisible WebView hijacking (IWH) attack to steal user credentials based on the proposed vulnerability. Any Android apps or third-party components that use WebView are able to launch this attack. In addition, it can be easily enabled in current popular apps. The key part of our proof-of-concept implementation is illustrated in Appendix A. It offers a general, stealthy approach, which generally works for all the pages without user awareness, for the adversary to steal user credentials from the Android WebView. It does not have the following two limitations which common phishing attacks have. First, they consider specific web pages which they can forge, but the ones accessed by users may not be involved.

TABLE I
PUBLIC METHODS OF THE INPUTCONNECTION INTERFACE [69]. THE FIRST COLUMN IS A TAG THAT WE SET IN EACH FUNCTION CALL MEDIATED BY OUR ICWRAPPER.

Tag	**Function**	**Called by**	**Description**
IC.create	onCreateInputConnection	IMM Client	Create a new input connection when an editor gets focus.
IC.commit	commitText	IMS	Commit text to the text box.
IC.action	performEditorAction	IMS	Have the editor perform an action. (e.g., a button is clicked to submit form.)
IC.finish	finishComposingText	IMS	Have the editor finish whatever composing text is currently active.
IC.key	sendKeyEvent	IMS	Send a key event to the editor
IC.region	setComposingRegion	IMS	Mark a certain region of text as composing text.
IC.composing	setComposingText	IMS	Replace the currently composing text with the given text.
IC.delete	deleteSurroundingText	IMS	Delete a region of texts.
IC.dumpText	getExtractedText	IMS	Retrieve the current text in the editor.
IC.beforeCursor	getTextBeforeCursor	IMS	Get characters of text before the current cursor position
IC.afterCursor	getTextAfterCursor	IMS	Get characters of text after the current cursor position

Second, they may make user awareness without showing genuine content given the input of user credentials. Although it can be addressed by man-in-the-middle attacks, much more effort is needed. Moreover, many tools and research studies have been proposed to address phishing attacks.

We develop a malicious app for the attack and note two things. First, the attack can also be carried out by third-party components which are used by benign apps. Second, we uploaded the IWH app to the Google Play Store and discovered that it could bypass the verification of the app security check. The verification result shows "Verified by Play Protect." For the security reason that the app may be accidentally downloaded, we canceled its publishing right after the upload completes.

The malicious app loads each page that requires user inputs using WebView, and meanwhile, hijacks its input channel to collect the user inputs. However, identifying user credentials from a set of user inputs is not straightforward. They can be a combination set of user inputs and actions (e.g., typing and deleting characters and moving cursors). It requires much more effort than what current tasks are done by the WebView and the IMS. The WebView prints the characters which are sent through the input channel on the current cursor position. The IMS takes care of two major tasks: sending the characters typed by the user and optimizing user typing actions based on the state/text in the current input field (e.g., word suggestions). They care about only the string which is being formed by the user, but not the final string in each input field, which is needed to identify user credentials.

We address the following issues in our attack design.

- How to group user inputs which belong to an IC instance and maybe the final string of a field?
- How to form a string generated by an IC instance given its user inputs? The string is built based on multiple operations, including typing, moving cursors, deleting, etc. It can be in plain text or password format.
- How to resolve the repetition condition that the user may move the focus to one input field more than once so that multiple instances are created for that field?
- How to localize each input string among multiple input fields? The web pages requiring user credentials usually have more than one input field.

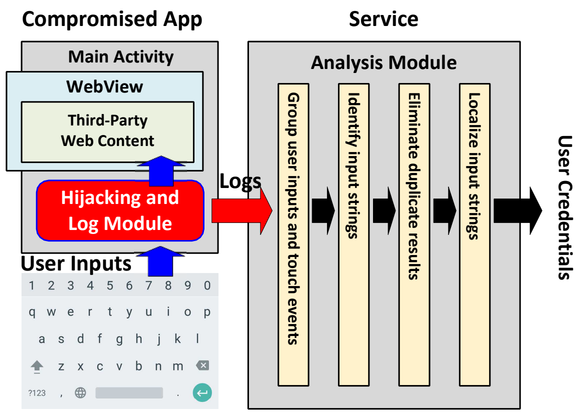


Fig. 7. The architecture of the IWH Attack.

Figure 7 shows the attack design, which mainly consists of two modules: hijacking/log and analysis modules. The hijacking/log module keeps monitoring user inputs through the ICWrapper and the touch events (e.g., Down and SingleTap) on the WebView from the Android API. The generated logs are sent to the analysis module, which runs as a local or remote service. The modularized design also makes it possible to minimize the footprint of the attack components by injecting only the hijacking and log module. In the analysis module, we do user input reconstruction as follows. We consider the string with which an IC instance ends up in the editor as a building block since the final result of each input field must come from one of these kinds of strings. To collect those strings, the app groups user inputs and touch events according to each IC instance and then identifies an input string from each group. It also eliminates duplicate results that exist for one input field and finally localizes collected input strings among the input fields. Finally, it can get a set of user credentials. The algorithm pseudocode of the user input extraction process is given in Algorithm 1. We elaborate on each component used in the algorithm below.

Grouping User Inputs and Touch Events. We assign a 32-bit identifier, which is randomly generated, to each IC instance in our ICWrapper middleware. It is attached to each log entry. For example, as shown in Figure 6, the identifier of the instance created for the username is `7ee6f925`, whereas that for the password is `4f1ec49f`. The log entries generated for the username/password inputs can be grouped based on the identifiers. Moreover, the group to which a touch event should belong is determined based on its identifier. We note two things. First, the ICWrapper can know when a new instance is created based on whether the `onCreateInputConnection` function is called. Second, we capture touch events by implementing `dispatchTouchEvent` and `GestureDetector` listeners in the WebView.

Identifying Input Strings. We next identify the final string of an IC instance based on its log entries. Due to different actions performed on non-password and password cases, we deal with them differently. Those two cases can be differentiated based on the information of the editor's input type. As shown in Figure 6, the input type of the username, which is plaintext, can be `WEB_EDIT_TEXT` or `WEB_EMAIL_ADDRESS`, whereas that of the password, which is presented by dots, is `WEB_PASSWORD`.

For the non-password case, we keep monitoring the editor's text throughout the lifetime of the IC instance. By following each call of IC functions from the IMS, our ICWrapper calls both functions of `getTextBeforeCursor` and `getTextAfterCursor` to obtain the whole text, which is the concatenation of the strings returned by those two functions. It is due to two reasons. First, the text change caused by the user's typing in an editor is done by the IC function calls. Second, whenever an instance is alive, there must be a cursor, which points out where newly typed characters should appear, in the editor. Once the observed instance identifier changes or the form is submitted, the latest observed text is considered an input field's final string.

However, the password string cannot be obtained by monitoring the editor's text because right after each character is delivered to the editor, it becomes a dot (•). We can only get a sequence of dots when calling the functions of getting the editor's text. We thus maintain intermediate states of user inputs in real-time to form the password string. We maintain a text buffer and a cursor position variable. When a character committed by the `commitText` function is observed, it is inserted to the current cursor position in the text buffer, and the position variable increases by one. We do deletion in the text buffer whenever any deletion event occurs. Note that We also need to monitor the current cursor position because the user may move the cursor by touching the view. Once a touch event is observed, we call the `getTextBeforeCursor` function to get the cursor position. Although this function returns only dots in this case, the cursor position can be inferred based on how many dots are in front of the cursor. Finally, the final password string can be extracted from the buffer.

Eliminating Duplicate Results. We detect duplicate results by checking whether the existing string of the input field on which the user newly focuses matches one of the input strings which we have identified. A match represents that the user is updating an input string that has been formed, so the old string will be replaced by the newly formed one. This way can prevent us from ending up with more than one input string belonging to an input field after all the inputs are done. To do the check, we obtain the existing string by calling the `getExtractedText` function right after the `onCreateInputConnection` function is called, and then compare it with the input strings which have been identified.

TABLE II
VARIOUS MOBILE DEVICES AND DIFFERENT DEVICE SETTINGS IN THE FIELD STUDY.

Mobile Devices		
Brand	**Keyboard**	**Models**
HTC	HTC Sense Input	ONE 801s, B830x, M10h, M9pw, U-2u
SAMSUNG	Samsung Keyboard	SM-G900F, SM-G930F
ASUS	ASUS Keyboard	Z00ED, Z012DA
LG	LG Keyboard	LGE Nexus 5
SONY	Xperia Keyboard	C6502, C6602, D6653, E2363, E6683, F5321, F8132, G3125, G8232
OPPO	Touchpal	A1601, F1f
Google	Gboard	Pixel 2
Android Emulator		

Android SDK Versions
4.4.2, 5.0, 5.0.2, 5.1, 5.1.1, 6.0, 6.0.1, 7.0, 7.1.1, 8.1.0, 9.0, 10.0, 11.0

Screen Resolutions (Pixel)
720x1280, 900x1600, 1080x1920, 1440x2560

DPI (Dots Per Inch)
240, 272, 320, 420, 480, 640

TABLE III
10 WEB PAGES USED IN THE FIELD STUDY.

#	Company	Category	Input Web Page
1-2	Amazon, Taobao	Online Retailer	Account Login
3	eBay	E-commerce	Account Login
4	Paypal	Online Payments	Account Login
5	Paypal	Online Payments	Credit Card Info.
6	American Express	Banking	Account Login
7-8	CapitalOne, Chase	Banking	Account Login
9-10	Facebook, Twitter	Social Media	Account Login

Localizing Input Strings. A web page may have multiple input fields, and they may be filled in any orders. The first three components only identify the strings inputted on the page but do not associate them with the input fields. This component is thus introduced to map the input strings to the input fields one by one. It does the mapping based on the orders of the input strings and fields on the page in terms of their vertical positions. The vertical position order of the input strings can be identified based on their absolute coordinates on the page, whereas that of the input fields can be learned by checking the page through a visited URL. We can obtain the coordinates of each input string based on its corresponding touch events, but they are relative coordinates of the current screen or display area. For one spot in a view area, the relative coordinate changes when the user scrolls the page, but the absolute one does not. We thus need to convert relative coordinates to absolute ones. We calculate the absolute coordinate of each touch event by getting both the scroll movement range (i.e., sx and sy) and the relative coordinate (i.e., tx and ty). The absolute coordinate can then be obtained by $(sx+tx, sy+ty)$. Given the absolute coordinate of each input string, the input strings can be ordered vertically and mapped to input fields one by one.

Algorithm 1 The user input extraction algorithm.

```
Require: Captured events $E = \{e_1, e_2, \ldots, e_n\}$ from a user
 1: $LD$ = {an empty dict contains lists of captured events}
 2: $R$ = {an empty list for storing extracted user inputs}
 3: // {Group User Inputs and Touch Events.}
 4: for all $e$ in $E$ do
 5:     Append $e$ to list $LD[e.id]$
 6: end for
 7: // {Identify Input Strings.}
 8: for all $l$ in $LD$ do
 9:     $s$ = {a buffer contains an empty string}
10:     $pos$ = 0 {cursor positon}
11:     $y$ = {an empty list for storing y coordinates}
12:     for all $e$ in $l$ do
13:         if $e.type$ in {beforeCursor, afterCursor} then
14:             Update $s$ based on $e$ and $pos$
15:         else if $e.type$ in {ComposingText, commitText} then
16:             Update $s$ and $pos$ based on $e$
17:         else if $e.type$ in {touchEvent} then
18:             Update $pos$ based on $e.x$
19:             Append $e.y$ to $y$
20:         else if $e.type$ in {finishComposing} then
21:             Replace $s$ based on $e$
22:         end if
23:     end for
24:     Append ($s$, $y$) to $R$
25: end for
26: // {Eliminate Duplicate Results.}
27: for all $r$ in $R$ do
28:     if $r.s$ is not unique in $R$ and $r.y$ then
29:         Remove $r$ from $R$
30:     end if
31: end for
32: // {Localize Input Strings}
33: Sort $r$ in $R$ based on $r.y$
34: return $R$
```

C. Field Study

We conduct a field study to examine whether our attack design can precisely acquire user credentials in most cases. We use our developed app for the study and ask volunteers from our department to test it[4]. We clone ten web pages containing input fields of login username/password or credit card information from popular Internet services. The cloned pages are from different companies in various industry categories, as shown in Table III. They do not provide any real services but are used for only the test of input data reconstruction; once a page is submitted, the browser either goes to the next test page or ends the test. Note that our server is the one to which these pages are submitted, so we can collect the actual user input of each test to gauge the accuracy.

For each test, the app randomly selects one from those ten pages for the tester to fill in the input fields. To emulate a real scenario of deployment such an attack, the app collects IMF and touch events on the device and periodically sends the collected events to a backend server, which runs our proposed algorithm to extract leaked user information. Since we examine whether any modification on the input data can be correctly detected by our algorithm, we request the tester to modify his/her input right after the first submission and then submit it again. Each test produces two test results with the same web page. We encourage testers to do multiple tests at their convenience. Overall, we collect 1537 test results from 90 testers. Their devices and settings are summarized in Table II.

We compare the actual user input with the output of our algorithm. It is observed that it can correctly identify user inputs in a total of 1507 tests and thus achieve 98.0% accuracy. 2.0% of errors come from the cases that the tester moves the cursor among characters within a password field by touching the screen and then modifies the password string. Our attack design may get the wrong locations of the cursor within the password string. This is mainly because the font size of an input field rendered on different devices could be diverse, and therefore coordinate-based cursor position prediction may still get an incorrect result even if we have considered DPI-based corrections on some devices. Moreover, the password string, each character of which is shown as a dot in the editor, cannot be obtained by monitoring the editor's text. This kind of modification can thus lead to identifying an incorrect password string. Totally, there are 38,839 input operations captured by the app. Among them, 4 percent of the operations are cursor movements, 4 percent are text deletions, and 92 percent are text composing operations.

Note that we did not collect the information of the Android patch levels and the WebView versions from the testers since most of them may have difficulty reporting it without a background in computer science. However, we have validated the vulnerability of the input channel hijacking on 22 device models across seven phone brands and Android versions from 4.4 to 11.0. We thus believe that the vulnerability is a common security issue of Android but not roots in some manufacturer customization.

VI. Solution

We devise two solutions for mobile web services and the Android platform, respectively, under the current IMF framework. We propose a *web-based virtual keyboard* for the web services to prevent users from using IMF for their inputs. The reasons are twofold. First, the vulnerability may not be fixed promptly by the Android team. Second, various phone models and Android versions may not be all updated to be fixed, even if an updated Android version is released. For the Android platform, we introduce an *IMF hijacking guardian* to set up a security association between the active IMS and the WebView's inherent IC instance (see reasons in the root cause of Section V-A).

A. Web-based Virtual Keyboard

The web-based virtual keyboard avoids using the system's input methods on acquiring user credentials. It is based on the emulation of a software keyboard using the `HTML span`

[4]The IRB of this field study has been approved.

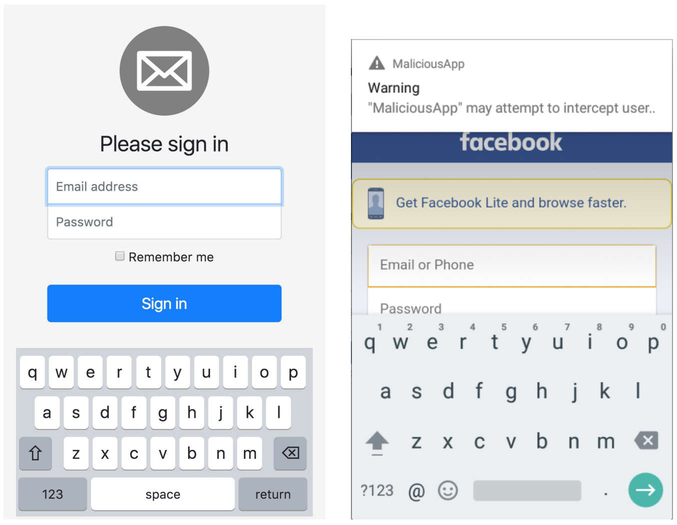


Fig. 8. Screenshots for the proposed solutions. Left: a sample web-based virtual keyboard shown on a web page; Right: a pop-up notification from the IMF hijacking guardian when an input hijacking occurs.

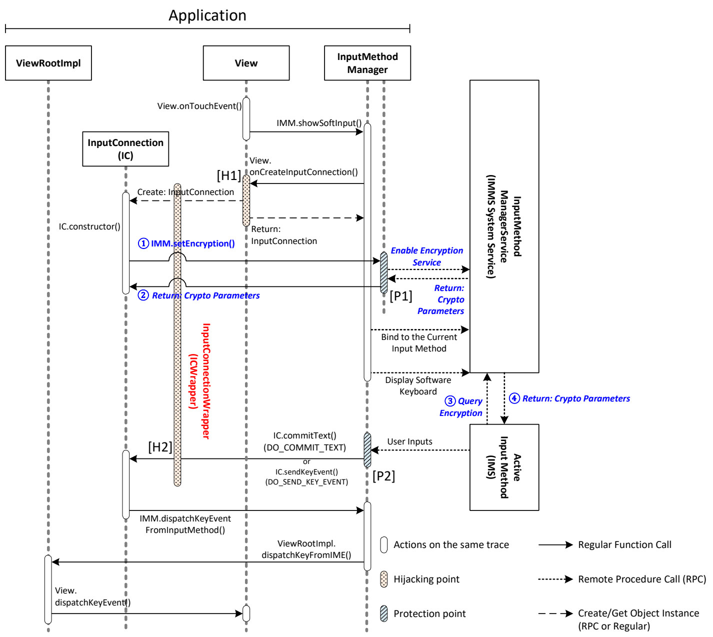

Application
ViewRootImpl
View
InputMethod Manager
InputConnection (IC)
InputMethod ManagerService (IMMS System Service)
View.onTouchEvent()
IMM.showSoftInput()
View. onCreateInputConnection()
[H1]
Create: InputConnection
Return: InputConnection
IC.constructor()
① IMM.setEncryption()
Enable Encryption Service
Return: Crypto Parameters
[P1]
② Return: Crypto Parameters
InputConnectionWrapper (ICWrapper)
Bind to the Current Input Method
Display Software Keyboard
③ Query Encryption
④ Return: Crypto Parameters
User Inputs
Active Input Method (IMS)
[H2]
IC.commitText() (DO_COMMIT_TEXT)
or
IC.sendKeyEvent() (DO_SEND_KEY_EVENT)
[P2]
IMM.dispatchKeyEvent FromInputMethod()
ViewRootImpl. dispatchKeyFromIME()
View. dispatchKeyEvent()
Actions on the same trace
Hijacking point
Protection point
Regular Function Call
Remote Procedure Call (RPC)
Create/Get Object Instance (RPC or Regular)

Fig. 10. Interactions between the IMF hijacking guardian, a view, and the IMF framework.

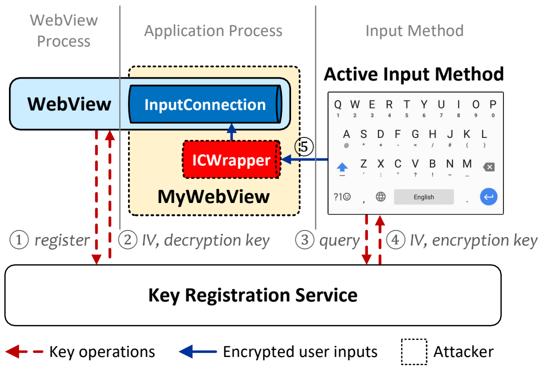


Fig. 9. The IMF hijacking guardian based on a security association between WebView's inherent IC instance and the active IMS.

element. The software keyboard is shown based on the `span` and hooked to enable user input with JavaScript codes. When any input field is clicked and thus becomes active, its hook will cause the virtual keyboard to show up.

Implementation. We carry out the following two major tasks for the virtual keyboard, as shown in the left of Figure 8. First, we set input fields to be *read-only* and draw the keyboard layout and keys using the `span`. The input field on which the user focuses is decorated with a different border color (e.g., blue in the sample). Second, the keys are hooked to append their corresponding characters to the focused input field when they are clicked, whereas the input fields are hooked to enable the appearance of the virtual keyboard. Since there is a focus on the field of "Email address," the virtual keyboard is shown; otherwise, it is hidden. It just works as normal input methods, which deliver characters one by one to an input field.

B. IMF Hijacking Guardian

This guardian prevents the hijacking by setting up a security association between WebView's inherent IC instance and the active IMS. The text sent from the IMS to the IC instance is encrypted by a symmetric key and a cryptographic algorithm, which are assigned by a key registration service in the Android system. It can enable the system to notify users of any suspicious signs that current foreground apps may do the hijacking. The signs allow users to consider to do input with other trustable apps. In addition, even if the input channel is hijacked, the encryption prevents user inputs from being leaked.

Figure 9 shows the guardian's operation. Whenever an editor is focused, the WebView registers the service and obtains a set of the key and algorithm. Afterward, the active IMS also fetches that security context so that it can encrypt the user's input text. The delivery of the encrypted text is shown with the solid blue arrow in the figure. Therefore, the malicious app that hijacks the input channel is not able to obtain the input text in plaintext.

Note that a malicious app may register its view to the key registration service and then obtain the security context shared with the current IMS. The user's input text can be decrypted only by the malicious app's customized IC. It results in that no text input appears on the wrapped view, and the user can thus be aware of the anomaly.

Implementation. We implement the IMF hijacking guardian in the framework of Android version 7.1 and generate a patched framework to run in an emulator to show its effectiveness. Figure 10 shows the detailed interactions between the IMF hijacking guardian, a view, and the IMF framework. We implement a key registration service as an additional feature in the original IMM service. To ensure compatibility for existing applications, the text input encryption feature is disabled by default in the constructor of an IC object. An IC held by sensitive views such as WebView can enable this feature by calling the `setEncryption` function, as `[P1]` shown in the figure. This function should be called every time an IC

TABLE IV
PERFORMANCE IMPACTS FOR TEXT INPUT ENCRYPTION AND DECRYPTION OPERATIONS.

	1-byte	2-byte	4-byte	8-byte	16-byte
AES-128	0.365ms	0.364ms	0.359ms	0.357ms	0.367ms
AES-192	0.349ms	0.356ms	0.361ms	0.363ms	0.352ms
AES-256	0.358ms	0.358ms	0.359ms	0.358ms	0.368ms
RC4	0.257ms	0.252ms	0.250ms	0.251ms	0.244ms

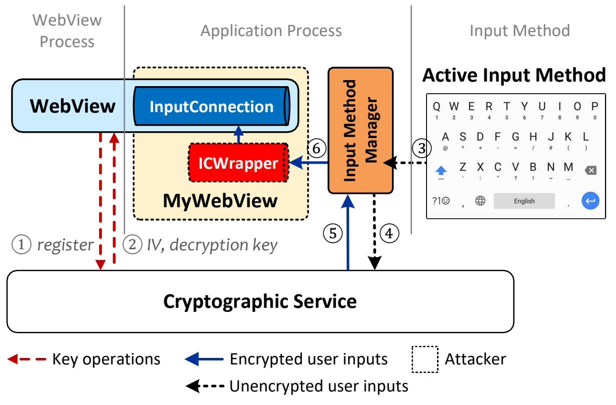
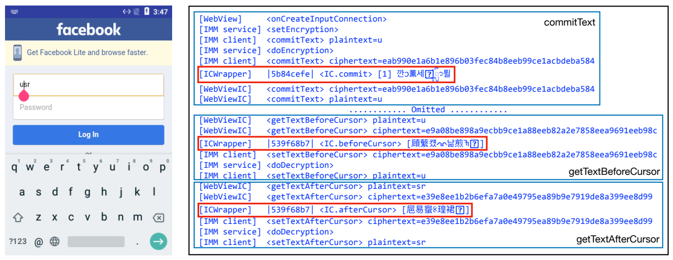


Fig. 11. The IMF hijacking guardian is provided from a standalone cryptographic service.

is instantiated or the `onCreateInputConnection` is called. The crypto parameter used for text encryption is thus generated and returned to the caller. The key registration service maintains only a single instance of crypto parameters. A second call to the `setEncryption` function generates a new instance of crypto parameters and replaces the existing one. Once the text encryption service is activated, an active IMS is able to retrieve the crypto parameters from the key registration service. Note that the key registration service only responds to the query coming from an active IMS; otherwise, an error code is returned.

With the correct crypto parameters, the text messages exchanged between the target IC held by a WebView and the active IMS are encrypted bidirectionally. The IMS can send encrypted user inputs to the IMM, and then the IMM forwards them to the IC via `commitText` and `setComposingText`. Similarly, the IC can encrypt all the responses to the queries, e.g., `getTextBeforeCursor` and `getTextAfterCursor`, from other components in the system, and only the active IMS can decrypt the responses. An anomaly is identified at the IC when the received messages cannot be decrypted correctly. The right side of Figure 8 shows an example that a warning notification from the IMF hijacking guardian is popped up on a malicious app that launches the IWH attack after the user sends a text input to an input field and the IC cannot correctly decrypt the input.

Performance Impact. We next examine the delays imposed by the text input encryption/decryption and see whether they can get user awareness. We conduct experiments on a moderate CPU (Qualcomm APQ8084) running at 1.5 GHz and emulate user inputs by generating 1-, 2-, 4-, 8-, and 16-byte texts. We consider four different cipher algorithms: AES-128, AES-192, AES-256, and RC4. In each test, there are 20,000 runs. At each run, the text is encrypted and then decrypted. The average delays are presented in Table IV. It is observed that the delays are less than 0.4 ms in all the cases. Such delays are considered negligible and are not noticeable to smartphone users, since it is even smaller than the fastest refresh interval of current smartphone displays. The refresh rates of the displays range from 60 Hz to 120 Hz; that is, the smartphone screens are redrawn with an interval from 16.67 ms to 8.33 ms.



```
[WebView]     <onCreateInputConnection>
[IMM service] <setEncryption>
[IMM client]  <commitText> plaintext=u
[IMM service] <doEncryption>
[IMM client]  <commitText> ciphertext=eab990e1a6b1e896b03fec84b8eeb99ce1acbdeba584
[ICWrapper]   |5b84cefe| <IC.commit> [1] 깐ↄ薫세?ↄ뢸
[WebViewIC]   <commitText> ciphertext=eab990e1a6b1e896b03fec84b8eeb99ce1acbdeba584
[WebViewIC]   <commitText> plaintext=u
             ............ Omitted ............
[WebViewIC]   <getTextBeforeCursor> plaintext=u
[WebViewIC]   <getTextBeforeCursor> ciphertext=e9a08be898a9ecbb9ce1a88eeb82a2e7858eea9691eeb98c
[ICWrapper]   |539f68b7| <IC.beforeCursor> [顚蘩갰ᨒ낢煎ʞ?]
[IMM client]  <setTextBeforeCursor> ciphertext=e9a08be898a9ecbb9ce1a88eeb82a2e7858eea9691eeb98c
[IMM service] <doDecryption>
[IMM client]  <setTextBeforeCursor> plaintext=u
[WebViewIC]   <getTextAfterCursor> plaintext=sr
[WebViewIC]   <getTextAfterCursor> ciphertext=e39e8ee1b2b6efa7a0e49795ea89b9e7919de8a399ee8d99
[ICWrapper]   |539f68b7| <IC.afterCursor> [屉易窒ৼ瑝裙?]
[IMM client]  <setTextAfterCursor> ciphertext=e39e8ee1b2b6efa7a0e49795ea89b9e7919de8a399ee8d99
[IMM service] <doDecryption>
[IMM client]  <setTextAfterCursor> plaintext=sr
```

Fig. 12. Demonstration of the alternative deployment that runs the encryption service in an isolated process: the hijacked channel can only receive encrypted input texts.

Alternative Deployment. The IMF hijacking guardian requires some new functions (e.g., text encryption) to be supported by input methods, but this requirement may be difficult for those developed by the third party. An alternative deployment approach is to run the encryption service in an isolated process or along with the IMM service, as shown in Figure 11. Upon the receipt of an input text at the IMM, it is forwarded to the encryption service via an interprocess communication channel and then encrypted. Afterward, the encrypted text is sent back to the IMM and then forwarded to the IC. A pitfall of this approach is that the input text without encryption needs to traverse the IMM in the app's main process. It leaves a chance for the adversary to steal the text by scanning memory. However, the chance is very small because the time interval of the traverse is very short. Figure 12 demonstrates the alternative deployment with the built-in input method in an Android emulator. It is observed that the ICWrapper can receive only encrypted text inputs.

VII. DISCUSSION

In this section, we first examine the IWH attack on password managers and then discuss several attacks which can be launched in the same threat model but have more limitations than IWH. Note that IWH can be applied to only Android, so we focus the discussion on the attacks against Android. We leave the examination of the related attacks on iOS to our future work.

IWH on Password Managers. We examine whether the IWH attack works for password managers [70] that use the Accessibility or Autofill services to fill the password field. We develop two password manager apps using the Accessibility and Autofill services, respectively, and experimentally test IWH on them. We discover that IWH does not work for them, but an approach similar to the InputConnection hijacking can be used to steal the password when the Autofill service is used. Specifically, an attacker can inherit the WebView class and do hijacking in the function used by the Autofill service to send a string to the password field. For the Accessibility service, although the IMF event can be triggered, the password captured from the IWH attack is just a string of dots, and thus the attack fails to obtain the password. However, it has been suggested that the Accessibility service should not be enabled for password manager apps since it gives much more privileges than those needed by the apps and can open the door to security threats [71].

Malicious Self-made Browser. To steal user credentials from the user input on an Android view, the adversary may use a self-made browser that does not inherit the WebView class; instead, the browser is directly built based on the view class. Without the restriction imposed by WebView, the adversary can easily obtain user credentials from the web pages rendered by the self-made browser. However, creating such a browser is challenging since the browser is a complex software component and requires great effort for a stable and standard-compliant version. Although there have been some open-source browser packages, the adversary still needs to patch and rebuild their source codes to enable the leakage of user credentials. Therefore, the attack based on the self-made browser needs a much higher implementation cost than the IWH attack and sets a high technical barrier for the adversary.

Moreover, embedding a self-made browser in a malicious app may increase the app's size to a huge number, which could be used as a feature for the detection of malicious apps by anti-virus software. The code size of an official WebView package ranges from 30 MB to 50 MB, so it is reasonable to estimate that the size of a self-made or open-source browser can be at least several tens MB. The size has been much larger than most apps, especially that the malicious app does not provide any multimedia functions. According to the statistics from two reports [72], [73], the average sizes of regular apps in 2012 and 2017 are only 6 MB and 15 MB, respectively. An app with a huge code size could be considered a suspicious app by the anti-virus software and perform further inspection.

Spoofed Login Forms. The adversary may simply spoof login forms of web services, but the spoofed forms can be easily noticed after the submission of user inputs. Although they can be used to successfully steal user credentials, the user can be aware of this phishing attack when no valid user information can be shown after a successful login, and then may take some preventive actions immediately (e.g., changing his/her password). IWH is much more threatening due to its stealthy eavesdropping, which does not affect the normal operation of a web service. As for conventional phishing and pharming attacks, no valid response can be shown after submission of user inputs in most cases so that the phishing or pharming pages can be easily suspected and detected. Moreover, the conventional attacks work for only some specific web services for which fake pages have been created in advance, but IWH can be performed on all the web services accessed through WebView.

Information Leakage from Memory. When the foreground app and its WebView object are in the same process, the adversary may be able to scan memory to steal user credentials by using native codes. However, since Android 8.0, the WebView has been isolated to another process from the app's main process.

JavaScript Injection Attacks. Attackers may inject JavaScript to extract the content of input fields from WebView, but it can be prevented based on the same-origin policy, where any JavaScript codes can only be injected into the main frame page rendered by WebView. Web servers can prevent injection attacks by embedding the input fields in non-main frame pages. However, our IWH attack can still work with that prevention manner. Moreover, our proposed virtual keyboard can also be made immune to the injection attacks by being embedded in a non-main frame page.

VIII. Conclusion

Embedding WebView is a common practice for modern Android apps that require showing users web pages. Users may input user credentials on the WebView for some login/payment pages, especially for popular OAuth pages. Such input data belonging to the page owners shall not be exposed to the foreground app. In this work, we discover that Android fails to protect the user's input data on the WebView from the app. It is caused by an Android design defect where the input channel between an editor and an active IMS can be easily hijacked. We devise an attack that can steal user inputs without user awareness and conduct a field study with our developed attack app. The result shows that the accuracy of successfully stealing user inputs can achieve 98.0%. We finally propose and prototype two solutions, the web-based virtual keyboard and the IMF hijacking guardian, for mobile web services and the Android system, respectively.

Our study yields two insights. First, two ends of a data flow, which may traverse insecure entities, shall set up an end-to-end security association. System designers should carefully examine whether each kind of data flows may be forced to traverse any suspicious entities or not. Second, given that two functions (the IC interface initialization and the IMS configuration in this work) are coupled together within a method when one of them has security issues, we may face the dilemma of disabling/modifying the method or adding security protection. The former may have compatibility issues with existing apps, so our solution takes the latter approach. We hope our study will raise awareness of this IWH security threat, and the insights will benefit the security enhancement of system designs.

Acknowledgement

This study is supported in part by Ministry of Science and Technology under grants number MOST 107-2221-E-009-028-MY3, MOST 109-2218-E-009-010, and MOST 109-2628-E-009-001-MY3, and in part by the Center for Open Intelligent Connectivity from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education in Taiwan. We would also like to thank anonymous reviewers for their constructive and insightful comments.

```java
public class PoCWebView extends WebView  {
    class MyICWrapper extends InputConnectionWrapper {
        private int id = -1; // IC instance id
        public int getid() { return id; }
        //// PoC: implement required constructors HERE
        ////      id is initialized with a random number.
        @Override
        public CharSequence getTextBeforeCursor(
                int n, int flags) {
            CharSequence cs =
                super.getTextBeforeCursor(n, flags);
            // PoC: log captured texts
            return cs;
        }
        @Override
        public CharSequence getTextAfterCursor(
                int n, int flags) {
            CharSequence cs =
                super.getTextAfterCursor(n, flags);
            // PoC: log captured texts
            return cs;
        }
        @Override
        public boolean setComposingText(
                CharSequence text, int newCursorPosition) {
            // Poc: log captured texts
            return super.setComposingText(text, newCursorPosition);
        }
        @Override
        public boolean commitText(
                CharSequence text, int newCursorPosition) {
            // PoC: log captured texts
            return super.commitText(text, newCursorPosition);
        }
        @Override
        public boolean finishComposingText() {
            // Poc: extract and log captured texts
            return super.finishComposingText();
        }
    }

    private MyICWrapper icw = null;

    //// PoC: implement required constructors HERE
    @Override
    public boolean dispatchTouchEvent(MotionEvent event) {
        // PoC: log captured touch event
        return super.onTouchEvent(event);
    }
    @Override
    public InputConnection onCreateInputConnection(
            EditorInfo outAttrs) {
        InputConnection ic =
            super.onCreateInputConnection(outAttrs);
        return ic == null ?
            null : (icw = new MyICWrapper(ic, false));
    }
}
```

Fig. 13. Proof-of-concept implementation of our hijacking and log module.

Appendix

A. Proof-of-Concept Implementation

We provide a proof-of-concept (PoC) implementation of our proposed hijacking and log module in Figure 13. A malicious developer can create a customized WebView class by inheriting the system WebView class and then overriding the `onCreateInputConnection` function. The overridden `onCreateInputConnection` function creates a customized `InputConnection` instance and intercept user inputs in the customized `InputConnection` instance. All user text inputs sent to the web page rendered by the customized WebView are captured by the customized InputConenction instance.

References

[1] StatCounter Global Stats, "Mobile and tablet internet usage exceeds desktop for first time worldwide," November 2016, http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide.

[2] K. Kloboves, "Continuing to make the web more mobile friendly," Google Webmaster Central Blog, March 2016, https://webmasters.googleblog.com/2016/03/continuing-to-make-web-more-mobile.html.

[3] J. Hong, "The State of Phishing Attacks," *Communications of the ACM*, vol. 55, no. 1, pp. 74–81, January 2012.

[4] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the App is That? Deception and Countermeasures in the Android User Interface," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2015.

[5] A. Oest, P. Zhang, B. Wardman, E. Nunes, J. Burgis, A. Zand, K. Thomas, A. Doupé, and G.-J. Ahn, "Sunrise to Sunset: Analyzing the End-to-end Life Cycle and Effectiveness of Phishing Attacks at Scale," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020, pp. 361–377.

[6] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic Pharming Attacks and Locked Same-origin Policies for Web Browsers," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 58–71.

[7] S. Marchal, K. Saari, N. Singh, and N. Asokan, "Know Your Phish: Novel Techniques for Detecting Phishing Sites and their Targets," *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems*, pp. 323–333, Jun. 2016.

[8] G. Ramesh, I. Krishnamurthi, and K. S. S. Kumar, "An efficacious method for detecting phishing webpages through target domain identification," *Decision Support Systems*, vol. 61, no. Supplement C, pp. 12 – 22, 2014.

[9] G. Bottazzi, E. Casalicchio, D. Cingolani, F. Marturana, and M. Piu, "MP-Shield: A Framework for Phishing Detection in Mobile Devices," in *Proceedings of the 3rd IEEE International Workshop on Cybercrimes and Emerging Web Environments*, 2015.

[10] L. Wu, X. Du, and J. Wu, "Effective Defense Schemes for Phishing Attacks on Mobile Computing Platforms," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 8, pp. 6678–6691, August 2016.

[11] C. Pham, L. A. T. Nguyen, N. H. Tran, E. Huh, and C. S. Hong, "Phishing-Aware: A Neuro-Fuzzy Approach for Anti-Phishing on Fog Networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1076–1089, 2018.

[12] O. K. Sahingoz, E. Buber, O. Demir, and B. Diri, "Machine learning based phishing detection from URLs," *Expert Systems with Applications*, vol. 117, pp. 345–357, 2019.

[13] G. Ho, A. Cidon, L. Gavish, M. Schweighauser, V. Paxson, S. Savage, G. M. Voelker, and D. Wagner, "Detecting and Characterizing Lateral Phishing at Scale," in *28th USENIX Security Symposium (USENIX Security 19)*, Aug. 2019, pp. 1273–1290.

[14] Y. Li, Z. Yang, X. Chen, H. Yuan, and W. Liu, "A stacking model using URL and HTML features for phishing webpage detection," *Future Generation Computer Systems*, vol. 94, pp. 27–39, 2019.

[15] A. Das, S. Baki, A. El Aassal, R. Verma, and A. Dunbar, "SoK: A Comprehensive Reexamination of Phishing Research From the Security Perspective," *IEEE Communications Surveys Tutorials*, vol. 22, no. 1, pp. 671–708, 2020.

[16] Anti-Phishing Working Group, "Phishing activity trends report: 4th quarter 2015," March 2016, https://docs.apwg.org/reports/apwg_trends_report_q4_2015.pdf.

[17] ——, "Phishing activity trends report: 4th quarter 2016," February 2017, https://docs.apwg.org/reports/apwg_trends_report_q4_2015.pdf.

[18] H. Wang, Y. Guo, Z. Ma, and X. Chen, "WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 71–82. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771795

[19] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-Party Libraries in Android Apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE-C, 2016.

[20] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "LibD: Scalable and Precise Third-Party Library Detection in Android Markets," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE, 2017.

[21] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting Third-Party Libraries in Android Applications with High Precision and Recall," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*, ser. SANER, 2018.

[22] K. Jain, "Warning: 18,000 android apps contains code that spy on your text messages," 2015, https://thehackernews.com/2015/10/android-apps-steal-sms.html.

[23] M. Kumar, "Backdoor in baidu android sdk puts 100 million devices at risk," 2015, https://thehackernews.com/2015/11/android-malware-backdoor.html.

[24] Facebook.com, "Success Stories," 2018, https://developers.facebook.com/success-stories/.

[25] S. Gastellier-Prevost, G. G. Granadillo, and M. Laurent, "A Dual Approach to Detect Pharming Attacks at the Client-Side," in *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security*, Feb 2011, pp. 1–5.

[26] Y. Cao, W. Han, and Y. Le, "Anti-phishing Based on Automated Individual White-list," in *Proceedings of the 4th ACM Workshop on Digital Identity Management*, 2008, pp. 51–60.

[27] G. Xiang, J. I. Hong, C. P. Rosé, and L. F. Cranor, "CANTINA+: A Feature-Rich Machine Learning Framework for Detecting Phishing Web Sites," *ACM Transactions on Information and System Security*, vol. 14, no. 2, pp. 21:1–21:28, 2011.

[28] T. Willis, "Policy and disclosure: 2020 edition," News and updates from the Project Zero team at Google, January 2000, https://googleprojectzero.blogspot.com/2020/01/policy-and-disclosure-2020-edition.html.

[29] D. Shin, H. Yao, and U. Rosi, "Supporting visual security cues for WebView-based Android apps," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013.

[30] E. Chin and D. Wagner, "Bifocals: Analyzing WebView Vulnerabilities in Android Applications," in *Proceedings of International Workshop on Information Security Applications*, 2013.

[31] J. Yu and T. Yamauchi, "Access Control to Prevent Attacks Exploiting Vulnerabilities of WebView in Android OS," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications and IEEE International Conference on Embedded and Ubiquitous Computing*, 2013, pp. 1628–1633.

[32] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[33] T. Luo, "Attacks and Countermeasures for WebView on Mobile Systems," Ph.D. dissertation, Syracuse University, 2014.

[34] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking Attacks on Web in Android, iOS, and Windows Phone," in *Proceedings of International Symposium on Foundations and Practice of Security*, 2012.

[35] H. Feng and K. G. Shin, "Understanding and Defending the Binder Attack Surface In Android," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 398–409.

[36] W. Diao, X. Liu, Z. Zhou, K. Zhang, and Z. Li, "Mind-Reading: Privacy Attacks Exploiting Cross-App KeyEvent Injections," in *Proceedings of the 20th European Symposium on Research in Computer Security*. Springer International Publishing, 2015, pp. 20–39.

[37] F. Mohsen and M. Shehab, "Android keylogging threat," in *Proceedings of the 9th International Conference Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2013.

[38] L. Cai and H. Chen, "TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Security*, 2011.

[39] J. Cho, G. Cho, and H. Kim, "Keyboard or keylogger?: A security analysis of third-party keyboards on Android," in *Proceedings of the 13th Annual Conference on Privacy, Security and Trust*, 2015.

[40] M. H. Ahmadzadegan, A. Khorshidvand, and M. Pezeshki, "A method for securing username and password against the Keylogger software using the logistic map chaos function," in *Proceedings of the 2nd International Conference on Knowledge-Based Engineering and Innovation*, 2015.

[41] T. Fiebig, J. Danisevskis, and M. Piekarska, "A Metric for the Evaluation and Comparison of Keylogger Performance," in *Proceedings of the 7th USENIX Workshop on Cyber Security Experimentation and Test*, 2014.

[42] E. Ulqinaku, L. Malisa, J. Stefa, A. Mei, and S. Capkun, "Using Hover to Compromise the Confidentiality of User Input on Android," in *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2017.

[43] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, "Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, November 2013, pp. 611–622.

[44] J. Cappos, L. Wang, R. Weiss, Y. Yang, and Y. Zhuang, "BlurSense: Dynamic Fine-Grained Access Control for Smartphone Privacy," in *Proceedings of 2014 IEEE Sensors Applications Symposium*, 2014.

[45] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:14.

[46] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "On the security and engineering implications of finer-grained access controls for android developers and users," in *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2015.

[47] D. Liu, "Enhanced Password Security on Mobile Devices," Ph.D. dissertation, Duke University, 2013.

[48] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox, "ScreenPass: Secure Password Entry on Touchscreen Devices," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[49] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[50] L. Zhang, Z. Cai, and X. Wang, "FakeMask: A Novel Privacy Preserving Approach for Smartphones," *IEEE Transactions on Network and Service Management*, vol. 13, no. 2, pp. 335–348, 2016.

[51] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing the Walking Dead: Understanding Cross-App Remote Infections on Mobile WebViews," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 829–844.

[52] M. G. S. Jana and V. Shmatikov, "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks," in *Proceedings of 2014 Network and Distributed System Security (NDSS '14)*, San Diego, February 2014.

[53] G. Yang, A. Mendoza, J. Zhang, and G. Gu, "Precisely and Scalably Vetting JavaScript Bridge In Android Hybrid Apps," in *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[54] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha, "Secure Integration of Web Content and Applications on Commodity Mobile Operating Systems," in *Proceedings of the 12th ACM on Asia Conference on Computer and Communications Security*, 2017.

[55] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A System for Uniform and Fine-grained Access Control for Web Code on Android," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 104–115.

[56] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 5:1–5:29, Jun. 2014.

[57] K. Fawaz, H. Feng, and K. G. Shin, "Anatomization and Protection of Mobile Apps' Location Privacy Threats," in *Proceedings of the 24th USENIX Security Symposium*. Washington, D.C.: USENIX Association, 2015, pp. 753–768.

[58] W. Koch, A. Chaabane, M. Egele, W. Robertson, and E. Kirda, "Semi-automated Discovery of Server-based Information Oversharing Vulnerabilities in Android Applications," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 147–157.

[59] M. Wu, R. C. Miller, and S. L. Garfinkel, "Do Security Toolbars Actually Prevent Phishing Attacks?" in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2006, pp. 601–610.

[60] L. Wu, X. Du, and J. Wu, "MobiFish: A lightweight anti-phishing scheme for mobile phones," in *Proceedings of the 23rd International Conference on Computer Communication and Networks*, 2014.

[61] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[62] S. Fahl, M. Harbach, M. Oltrogge, T. Muders, and M. Smith, "Hey, You, Get Off of My Clipboard-On How Usability Trumps Security in Android Password Managers," in *Proceedings of Financial Cryptography and*

Data Security, A.-R. Sadeghi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 144–161.

[63] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan, “You Shouldn’t Collect My Secrets: Thwarting Sensitive Keystroke Leakage in Mobile IME Apps,” in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[64] B. Andow, A. Acharya, D. Li, W. Enck, K. Singh, and T. Xie, “UiRef: Analysis of Sensitive User Inputs in Android Applications,” in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Jul. 2017.

[65] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, “SUPOR: Precise and Scalable Sensitive User Input Detection for Android Apps,” in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC’15. Berkeley, CA, USA: USENIX Association, 2015, pp. 977–992. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831205

[66] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, “UIPicker: User-Input Privacy Identification in Mobile Applications,” in *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 993–1008.

[67] Android.com, “InputMethodManager API Reference,” 2018, https://developer.android.com/reference/android/view/inputmethod/InputMethodManager.html.

[68] ——, “Android 8.0 behavior changes,” https://developer.android.com/about/versions/oreo/android-8.0-changes.html#security-all, 2018.

[69] ——, “InputConnection API Reference,” 2018, https://developer.android.com/reference/android/view/inputmethod/InputConnection.html.

[70] S. Aonzo, A. Merlo, G. Tavella, and Y. Fratantonio, “Phishing Attacks on Modern Android,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1788—-1801.

[71] W. Diao, Y. Zhang, L. Zhang, Z. Li, F. Xu, X. Pan, X. Liu, J. Weng, K. Zhang, and X. Wang, “Kindness is a Risky Business: On the Usage of the Accessibility APIs in Android,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 261–275.

[72] ABI Research, “Average Size of Mobile Games for iOS Increased by a Whopping 42% between March and September,” ABI Research Press, Oct. 2012, https://www.abiresearch.com/press/average-size-of-mobile-games-for-ios-increased-by-/.

[73] B. Boshell, “Average app file size: Data for Android and iOS mobile apps,” Sweet Pricing Blog, Feb. 2017, https://sweetpricing.com/blog/2017/02/average-app-file-size/.

Chi-Yu Li is currently an Associate Professor with the Department of Computer Science, National Yang Ming Chiao Tung University (NYCU). He received the Ph.D. degree in computer science from the University of California, Los Angeles (UCLA) in 2015. His research interests include wireless networking, mobile networks and systems, and network security. He was awarded MTK Young Chair Professor in 2016.

Hsin-Yi Wang is currently a production engineer in Verizon Media, Inc. She received an M.S. degree in computer science from National Chiao Tung University (NCTU) under the supervising of Prof. Chun-Ying Huang. Before joining NCTU, she received her Bachelor’s degree from the Department of Computer Science and Engineering, National Taiwan Ocean University. Her research interests include user privacy, system security, and network security.

Wei-Ching Wang is currently a software engineer in Trend Micro, Inc. He received an M.S. degree in computer science from National Chiao Tung University (NCTU) under the supervising of Prof. Chun-Ying Huang. Before joining NCTU, he received his Bachelor’s degree from the Department of Computer Science, National Tsing Hua University. His research interests include user privacy, web security, and mobile security.

Chun-Ying Huang is a Professor at the Department of Computer Science, National Yang Ming Chiao Tung University (NYCU). Dr. Huang leads the security and systems laboratory in NYCU. He received the Ph.D. in Electrical Engineering from National Taiwan University in 2007. His research interests include system security, multimedia networking, and mobile computing. Dr. Huang is a member of ACM and IEEE. He was awarded ACM Taipei/Taiwan Chapter K. T. Li Young Researcher Award in 2014.