Recommended Paper

Access Control Mechanism to Mitigate Cordova Plugin Attacks in Hybrid Applications

Naoki Kudo[1] Toshihiro Yamauchi[1,a)] Thomas H. Austin[2,b)]

Received: April 30, 2017, Accepted: February 1, 2018

Abstract: Hybrid application frameworks such as Cordova are more and more popular to create platform-independent applications (apps) because they provide special APIs to access device resources in a platform-agonistic way. By using these APIs, hybrid apps can access device resources through JavaScript. In this paper, we present a novel *app-repackaging attack* that repackages hybrid apps with malicious code; this code can exploit Cordova's plugin interface to steal and tamper with device resources. We address this attack and cross-site scripting attacks against hybrid apps. Since these attacks need to use plugins to access device resources, we refer to both of these attacks as *Cordova plugin attacks*. We further demonstrate a defense against *Cordova plugin attacks* through the use of a novel runtime access control mechanism that restricts access based on the mobile user's judgement. Our mechanism is easy to introduce to existing Cordova apps, and allows developers to produce apps that are resistant to *Cordova plugin attacks*. Moreover, we evaluate the effectiveness and performance of our mechanism.

Keywords: hybrid Application, Android, Access Control

1. Introduction

In developing mobile applications (apps), hybrid apps are more and more popular because they can access the device resource in a platform-independent way. Unlike conventional mobile apps, hybrid apps are largely implemented using platform-independent languages such as HTML and JavaScript, with minimal use of platform-dependent languages such as Java on Android or Objective-C and Swift on iOS. Thus, a major advantage of hybrid apps is that mobile developers can share source code among different platforms. In addition, hybrid apps execute within WebView for using HTML and JavaScript.

Hybrid apps can access device resources through JavaScript by using a bridge that communicates between JavaScript code and platform-dependent language code. Hybrid apps are typically developed using hybrid application frameworks such as Cordova [2]. According to Android apps statistics [3], 6.3% of Android apps are implemented by using Cordova. Cordova apps use plugins as interfaces to access device resources.

In this paper, we present a novel *app-repackaging attack* that repackages Cordova apps with malicious code. App-repackaging attacks can steal and tamper with device resources by exploiting Cordova's plugin interface. In addition to these attacks, we address cross-site scripting attacks against hybrid apps [1]. We refer to these attacks as *Cordova plugin attacks* since they need to use plugins to access device resources. To address *Cordova plugin attacks*, we propose an access control mechanism that restricts access at runtime based on the mobile user's judgement.

Several works have introduced more fine-grained access control mechanisms in hybrid apps such as NoFrak [4], Jin et al. [5], and Mohamed et al. [6]. None of the previous researches considered access control based on a mobile user's judgement. In contrast, MobileIFC [7] proposes an access control mechanism based on the mobile user's judgement. However, MobileIFC is difficult to introduce to existing Cordova apps. On the other hand, the proposed technique can control access to device resources for plugins based on the mobile user's judgement at runtime, and can easily be applied to existing Cordova apps. Using our technique, it is possible to use Cordova apps more safely. Note: In this study, we focused on the Cordova framework for Android.

The contributions of this paper are as follows:

- We present a novel *app-repackaging attack* that repackages Cordova apps with malicious code. Malicious attackers can inject JavaScript code into existing Cordova apps. Moreover, *app-repackaging attacks* are more vulnerable to this form of code injection than Android apps. Therefore, this attack represents a significant threat because attackers can inject any code more easily.
- We propose an access control mechanism that restricts access to device resources based on the user's judgement for mitigating *Cordova plugin attacks*. App developers can easily introduce our mechanism since they do not need to modify the app's source code.

[1] Graduate School of Natural Science and Technology, Okayama University, Okayama 700–8530, Japan
[2] San Jose State University, San Jose, USA
a) yamauchi@cs.okayama-u.ac.jp
b) thomas.austin@sjsu.edu

The preliminary version of this paper was presented at Computer Security Symposium 2016 (CSS2016) in October 2016, and recommended to be submitted to Journal of Information Processing (JIP) by the program chair of CSS2016.

2. Cordova Apps

2.1 Structure of Cordova Apps

2.1.1 Structure

Figure 1 shows the structure of Cordova apps on Android.

Cordova apps use WebView and a Cordova framework. WebView shows web pages used by HTML and JavaScript. The Cordova framework helps app developers to develop Cordova apps by using HTML and JavaScript. As shown in Fig. 1, Cordova apps can access device resources by using plugins. By using plugins, these apps can access device resources across different platforms, such as iOS and Windows Phone. Cordova apps access device resources as follows:

(1) The Cordova app accesses the Java plugin from the JavaScript plugin.
(2) The Cordova app accesses device resources from the Java plugin.
(3) The Java plugin receives the result of the accessing device resources.
(4) The JavaScript plugin receives the result of accessing device resources from the Java plugin.

2.1.2 Plugins

A plugin is an interface to access device resources, and is divided into two parts: a JavaScript plugin and a Java plugin. The JavaScript plugin defines JavaScript APIs to access Java methods, while the Java plugin defines Java methods, which can access device resources. Cordova apps can access device resources through the JavaScript code by using JavaScript APIs.

Plugins are divided into two classes depending on providers. One is a Cordova core plugin provided by Apache Cordova and the other is a third party plugin provided by third parties.

2.2 Flow of Access to Device Resources for Plugins

Figure 2 shows a flow of access to device resources for plugins. Cordova apps access device resources through JavaScript as follows:

(1) Cordova determines whether the JavaScript code calls a JavaScript method for the plugin.
 (A) When the JavaScript code calls the JavaScript method for the plugin, the Cordova app starts the JavaScript plugin execution and calls the JavaScript API.
 (B) When the JavaScript code does not call the JavaScript method for the plugin, the JavaScript code completes execution.
(2) Cordova determines whether the URL is whitelisted.
 (A) When the URL is whitelisted, the Cordova app starts the Java plugin execution and calls the Java method corresponding to the JavaScript API.
 (B) When the URL is not whitelisted, the JavaScript plugin completes execution.
(3) The Cordova app accesses device resources through the Java method.
(4) The Java plugin sends the result of accessing device resources to the JavaScript plugin and then the Java plugin and the JavaScript plugin complete execution.

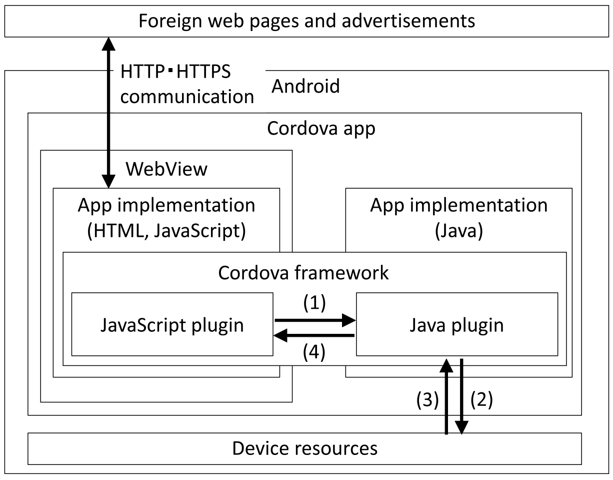

Foreign web pages and advertisements
HTTP・HTTPS communication
Android
Cordova app
WebView
App implementation (HTML, JavaScript)
App implementation (Java)
Cordova framework
(1)
JavaScript plugin
Java plugin
(4)
(3)
(2)
Device resources

Fig. 1 Structure of Cordova apps on Android.

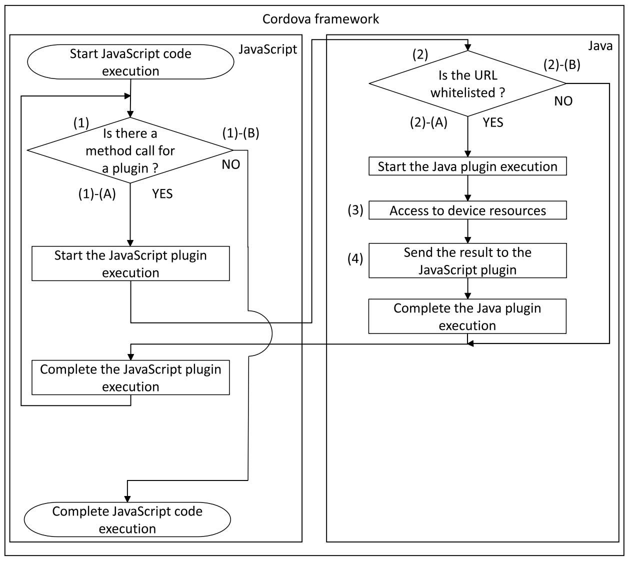

Cordova framework
JavaScript
Java
Start JavaScript code execution
(1)
Is there a method call for a plugin ?
(1)-(B)
NO
(1)-(A)
YES
Start the JavaScript plugin execution
Complete the JavaScript plugin execution
Complete JavaScript code execution
(2)
Is the URL whitelisted ?
(2)-(B)
NO
(2)-(A)
YES
Start the Java plugin execution
(3)
Access to device resources
(4)
Send the result to the JavaScript plugin
Complete the Java plugin execution

Fig. 2 Flow of access to device resources for plugins in the Cordova framework.

2.3 Problem of Cordova Apps

By using plugins, Cordova apps can access device resources through JavaScript. Therefore, Cordova apps can easily use device resources across different platforms such as iOS and Windows Phone by using plugins. However, when malicious attackers exploit plugins, they can steal and tamper with device resources through JavaScript.

3. Cordova Plugin Attacks

3.1 Threat Model

Although their ability to attack is limited to plugins read by the Cordova app, when malicious attackers exploit Cordova's plugin interface, they can steal and tamper with device resources through JavaScript as mentioned in Section 2.3. In addition, Jin et al. [1], Mohamed et al. [6], and Brucker et al. [8] show that malicious attackers can steal and tamper with device resources by exploiting plugins. Therefore, app developers need to address this problem to protect device resources from attackers. We analyze the structure of plugins and find a novel *app-repackaging attack* that injects malicious code into Cordova apps. Two forms of code injection attacks are focused on in this paper:

```
01: document.addEventListener('deviceready',
        this.getContacts, false);
02: getContacts: function () {
03:   receivedEvent('deviceready');
04:   function onSuccess(con) {
05:     for (var i=0; i<con.length; i++) {
06:       for (var j=0; j<con[i].emails.length; j++) {
07:         m = 'Name:'+con[i].name.formatted+
                'Address:'+con[i].emails[j].value;
08:         alert(m);
09:         b = document.createElement('img');
10:         b.src = 'http://127.***.**.**?m='+m; }}};
11:   function onError(contactError) {
12:       alert('onError');
13:   };
14:   var options = new ContactFindOptions();
15:   options.filter = "";
16:   options.multiple = true;
17:   options.hasPhoneNumber = true;
18:   var fields = [navigator.contacts.fieldType.name];
19:   navigator.contacts.find(fields, onSuccess,
          onError, options); }
```

Fig. 3 Injected JavaScript code by using an app-repackaging attack.

(1) App-repackaging attack
Malicious attackers can inject JavaScript code by repackaging Cordova apps, which are more vulnerable to this form of code injection than Android apps. *App-repackaging attacks* are significant threats because attackers can inject any code more easily.

(2) Cross-site scripting attack
Jin et al. [1] demonstrate that hybrid apps including Cordova apps have broad attack surfaces such as Wi-Fi access points and 2D barcodes, and malicious attackers can inject the code by using cross-site scripting vulnerabilities.

Furthermore, our *app-repackaging attack* significantly simplifies the problem of attacking an Android app. Typically, to modify Java bytecodes and repackage apps, attacking an app would involve time-consuming analysis of the target app and careful bytecode manipulation. However, since we are targeting hybrid applications, we only need to inject malicious JavaScript code into HTML. Therefore, we believe that *app-repackaging attacks* are easily and automatically applicable, and thus a serious threat.

We refer to both of these attacks as *Cordova plugin attacks*, since they leverage the Cordova's plugin interface. Note that in *Cordova plugin attacks* we focus on the problem that malicious attackers can inject the JavaScript code exploiting Cordova's plugins to access device resources.

3.2 Attack Example

To reveal threats of *Cordova plugin attacks*, we use a test app that we developed as an example. This app displays the Apache Cordova's webpage and accesses the InAppBrowser plugin and the Contacts plugin. Moreover, the test app has a vulnerability against *app-repackaging attacks*. We injected the JavaScript code of **Fig. 3** into the test app by using an *app-repackaging attack*.

The code of Fig. 3 uses `Contacts.find()` to get the user's

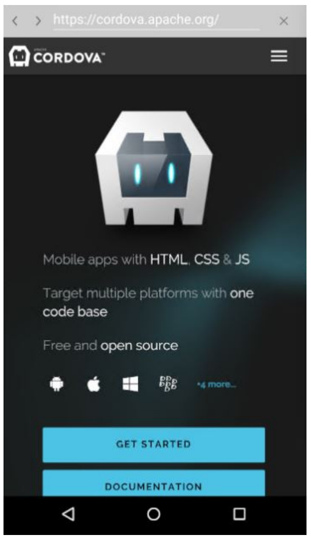
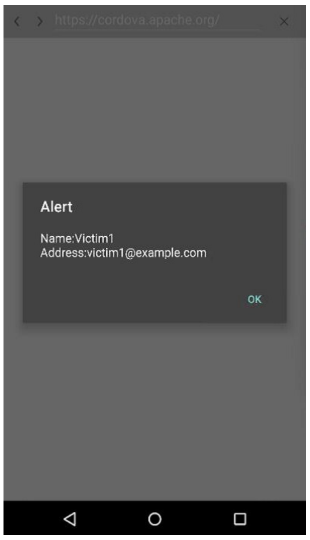

https://cordova.apache.org/
CORDOVA
Mobile apps with HTML, CSS & JS
Target multiple platforms with one code base
Free and open source
+4 more...
GET STARTED
DOCUMENTATION

Fig. 4 Screenshot of the original test app.



Alert

Name:Victim1
Address:victim1@example.com

OK

Fig. 5 Screenshot of the injected test app.

contacts by accessing the Contact plugin. The plugin API is called by the injected Cordova app in the case of the app startup. At Line 19, the plugin API gets contacts satisfying the conditions set by the code from Line 14 to Line 18. Then, the contacts are stored in the variable `con` and displayed at Line 08. At Line 09 and Line 10, the value of `con` is sent to the outside of the app (IP address `127.***.**.**`).

In addition, **Fig. 4** shows the screenshot of the original test app and **Fig. 5** shows the screenshot of the injected test app. In Fig. 5, the injected test app displays a dialog including mobile user's contacts before displaying the Apache Cordova's webpage. The operation of the injected test app corresponds to the code at Line 09 of Fig. 3. Moreover, we confirmed that the mobile user's contacts are sent to the outside. Thus, it shows that the JavaScript code shown in Fig. 3 is executed in the injected test app.

In this way, when Cordova apps have a vulnerability against *Cordova plugin attacks*, attackers can inject the JavaScript code for stealing and tampering with device resources into the Cordova apps.

3.3 App-Repackaging Attack

3.3.1 Structure of Apk Files for Android Cordova Apps

Before explaining how to inject the JavaScript code by repackaging Cordova apps, we show the architecture of apk files on Android Cordova apps in **Fig. 6**. The app source code of HTML and JavaScript are stored in /assets/www/*.

3.3.2 How to Inject JavaScript by Repackaging Cordova Apps

The process of injecting the JavaScript code by repackaging Cordova apps takes place as follows:

(1) Extracting apk files.
(2) Injecting the malicious JavaScript code into index.html or js files of /assets/www/.
(3) Repackaging apk files.

When malicious attackers exploit Cordova plugins to inject the JavaScript code, they can steal and tamper with device resources.

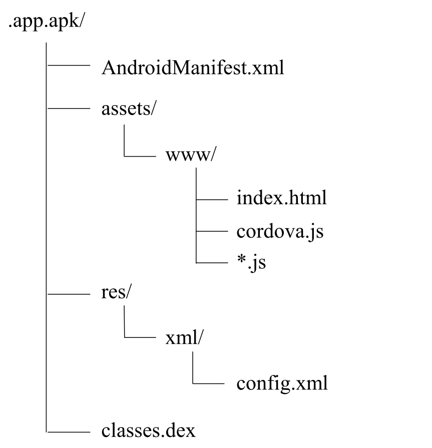


Fig. 6 Structure of apk files for Android Cordova apps.

3.3.3 Comparison with Repackaging Android Apps

Android malware using repackaging has increased in Android markets. Attackers repackage popular original Android apps in Google Play and spread repackaged Android apps in third party markets.

When attackers inject malicious code by repackaging Android apps, they exploit Java bytecode in class files extracted from a classes.dex file by using reverse engineering tools such as dex2jar [9] and Java Decompiler [10]. Moreover, they disassemble a classes.dex file into readable text files to know the application's operation. Therefore, repackaging Android apps takes time and effort to exploit Java bytecode and know the application's operation. In addition, when mobile developers use tools such as ProGuard [11] to obfuscate the Java code on Android apps, repackaging Android apps becomes difficult for attackers because attackers cannot know the application's operation exactly.

In contrast, when attackers inject malicious code by repackaging Cordova apps, they exploit the JavaScript code in /assets/www/ such as index.html and js files. Unlike repackaging Android apps, attackers can read the raw source code of these files directly. Moreover, since Cordova apps are typically written in HTML and JavaScript, the standard code obfuscation tools on Android apps are useless. In addition, even if mobile developers use JavaScript obfuscation tools on Cordova apps, attackers can inject the JavaScript code easily because these tools cannot obfuscate the source code written in HTML such as HTML tags.

Therefore, Cordova apps are more vulnerable to repackaging attacks than Android apps. In consequence, Cordova apps need a strong defense to mitigate the *app-repackaging attack*.

3.3.4 Comparison with Code Injection Attacks against Cordova Apps

In addition to *Cordova plugin attacks*, two code injection attacks exist: *Java bytecode injection attack* and *JavaScript code injection attack*. However, these attacks are not as serious as the Cordova plugin attacks. First, in a *Java bytecode injection attack*, attackers inject the exploit Java bytecode in class files extracted from a classes.dex file by using the same reverse engineering tools used in repackaging Android apps. In this attack, attackers can steal and tamper with device resources directly, despite the fact that this attack is a difficult one for attackers because they must precisely know the application's operations.

Next, in a *JavaScript code injection attack*, attackers exploit the JavaScript code in /assets/www/ such as index.html and js files. This attack can inject the JavaScript code easily, despite the fact that attackers cannot steal and tamper with device resources directly.

Compared to these attacks, *Cordova plugin attacks* are more serious because attackers can inject the code easily and can steal device resources directly.

3.4 Cross-Site Scripting Attack on Cordova Apps

In this section, we explain cross-site scripting attacks demonstrated in Jin et al. [1]. In a typical cross-site scripting attack, attackers inject the JavaScript code into the data field (such as in a form). Since the applications only interact with web servers, attackers use the site for their code to reach the victim's browser. On the other hand, hybrid apps have a much broader attack surface than web applications because they interact with many forms of entities, such as other apps, 2D barcode, Wi-Fi access points, other mobile devices, data sent by others or downloaded from external resources, etc.

Therefore, attackers can use many forms of entities to inject the JavaScript code compared to web applications. In one example, Jin et al. inject HTML tags and the JavaScript code into an existing hybrid app by using QR code and steal a device's geolocation.

3.5 Discussion

In this section, we consider whether the conventional Android system permission can protect device resources against *Cordova plugin attacks*. When Cordova apps access device resources by using plugins, they request Android permissions. Prior to Android 6.0, Android apps requested permissions at install-time. Since Android 6.0, Android apps request any permissions belonging to the "Dangerous Permissions" group at runtime. Requesting permissions at install-time cannot protect device resources against *Cordova plugin attacks* because it cannot detect access to device resources at runtime. On the other hand, requesting permissions at runtime can protect device resources against *Cordova plugin attacks* because it can detect access to device resources before plugins access them. Therefore, since Android 6.0, Cordova apps can prevent malicious JavaScript from accessing device resources for plugins belonging to the "Dangerous Permissions" group.

However, from Android 6.0 upwards, Android apps must set the targetSdkVersion to 23 or over for requesting permissions at runtime in the AndroidManifest.xml. According to Mutchler et al. [12], 93% of 60,086 Android apps had set the targetSdkVersion to under 23. Moreover, the attacker could change the targetSdkVersion's value to under 23 in order to facilitate *Cordova plugin attacks*. Therefore, it is assumed that many Cordova apps request permissions at install-time but not at runtime because they set the targetSdkVersion to under 23.

Consequently, many Cordova apps are vulnerable to *Cordova plugin attacks* because they request Android permissions at install-time. Therefore, Cordova apps need a strong defense to protect device resources from *Cordova plugin attacks*.

4. Access Control Mechanism for Plugins

4.1 Concept of Proposed Technique

To mitigate *Cordova plugin attacks* as described in Section 3.1, we propose an access control mechanism that restricts access to plugins before accessing and sending device resources at runtime. The purpose of the proposed technique is to prevent the malicious JavaScript code from exploiting Cordova's plugins to steal and tamper with device resources. We focus on the communication between a JavaScript plugin and a Java plugin into the Cordova framework as shown in Fig. 1.

Our access control mechanism can control the following.

(1) Access to the Java plugin from the JavaScript plugin before accessing device resources
(2) Information sent to the JavaScript plugin from the Java plugin after accessing device resources

By introducing the proposed technique, when mobile users use a vulnerable Cordova app, they can control access to device resources for plugins before accessing and sending device resources against *Cordova plugin attacks*.

Our access control mechanism can address *Cordova plugin attacks*. Moreover, app developers can easily integrate the technique into existing Cordova apps since they do not need to modify the app's source code.

4.2 Design

4.2.1 Access Control to the Java Plugin

Figure 7 shows an overview of access control to the Java plugin, which controls access to device resources from the JavaScript code as follows:

(1) A Java method that accesses the Java plugin is hooked.
(2) Our mechanism collects the necessary information from the method.
(3) Our mechanism determines whether access to this plugin has been granted previously.
    (A) When access to this plugin has not been granted, a dialog to decide the plugin permission is displayed.
    (B) When access to this plugin has been granted, the Cordova app starts the Java plugin execution and accesses device resources.
(4) Our mechanism controls this plugin according to the mobile user's judgement.
    (A) When the mobile user grants access to device resources, the Cordova app starts the Java plugin execution and accesses device resources.
    (B) When the mobile user denies access to device resources, the JavaScript plugin completes execution.

4.2.2 Information Control to the JavaScript Plugin

Figure 8 shows an overview of information control to the JavaScript plugin, which controls information sent to the JavaScript code as follows:

(1) A Java method that sends the result of accessing device resources to the JavaScript plugin is hooked.
(2) Our mechanism collects the necessary information from the method.
(3) Our mechanism determines whether sending to this plugin has been granted previously.
    (A) When sending to this plugin has not been granted, a dialog to decide the plugin permission is displayed.
    (B) When sending to this plugin has been granted, the Java plugin sends the result to the JavaScript plugin.
(4) Our mechanism controls this plugin according to the mobile user's judgement.
    (A) When the mobile user grants sending the result to the JavaScript plugin, the Java plugin sends the result to the JavaScript plugin.
    (B) When the mobile user denies sending the result to the JavaScript plugin, the control is moved to (5).
(5) Our mechanism restricts the information sent to the JavaScript plugin so as not to include device resources.

4.3 Challenges

To implement the proposed technique, we need to consider the following challenges.

C1 Controlling access to device resources for plugins based on the mobile user's judgement.
In vulnerable Cordova apps, the proposed technique needs to prevent the JavaScript code from accessing device resources through the Cordova plugin interface.

C2 Considering information that the dialog displays to the mobile user.
The proposed technique displays a dialog for access control based on the mobile user's judgement. The mobile user decides whether the Cordova app accesses and sends device

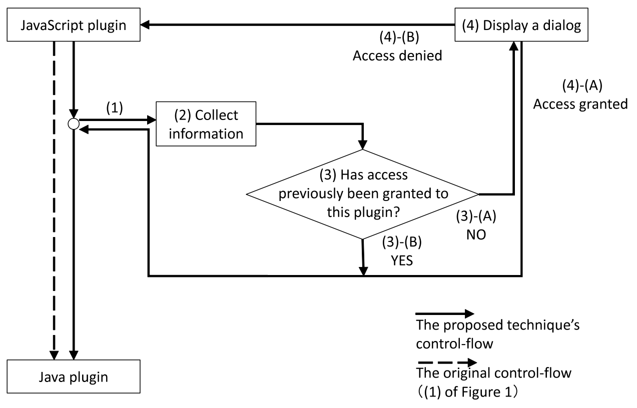

JavaScript plugin
(4) Display a dialog
(4)-(B)
Access denied
(4)-(A)
Access granted
(1)
(2) Collect information
(3) Has access previously been granted to this plugin?
(3)-(A)
NO
(3)-(B)
YES
The proposed technique's control-flow
The original control-flow ((1) of Figure 1)
Java plugin

Fig. 7 Overview of access control to the Java plugin.

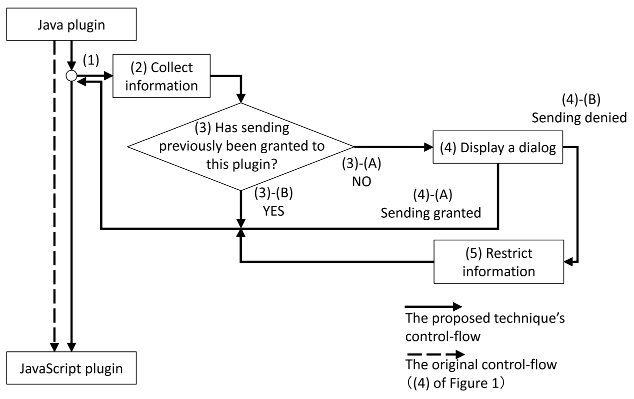


Fig. 8 Overview of information control to the JavaScript plugin.

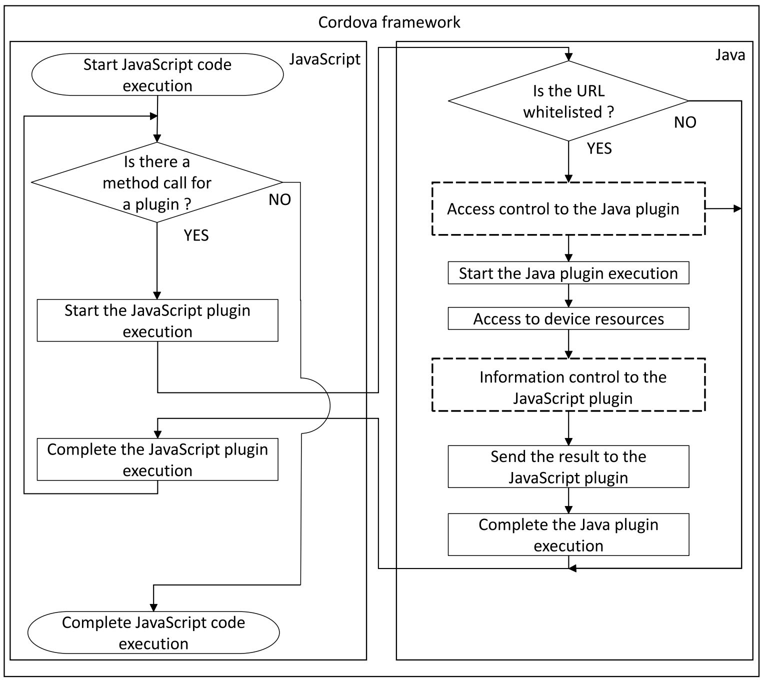

Cordova framework
JavaScript
Start JavaScript code execution
Is there a method call for a plugin ?
YES
NO
Start the JavaScript plugin execution
Complete the JavaScript plugin execution
Complete JavaScript code execution
Java
Is the URL whitelisted ?
NO
YES
Access control to the Java plugin
Start the Java plugin execution
Access to device resources
Information control to the JavaScript plugin
Send the result to the JavaScript plugin
Complete the Java plugin execution

Fig. 9 Flow of access to device resources for a plugin using the proposed technique in the Cordova framework.

resources for the plugin based on the information of the dialog.

C3 Avoiding repeated dialogs.
Once the user has granted access to a resource for a plugin, the plugin is assumed to retain that permission going forward. This design avoids excessive dialog messages that inconvenience for the mobile user.

4.4 Our Solution

4.4.1 Controlling Access to Device Resources for Plugins Based on the Mobile User's Judgement

The proposed technique displays a dialog based on the plugin name extracted from the hooked Java method. The mobile user decides whether the Cordova app accesses and sends device resources based on the information in the dialog. When the mobile user denies access to device resources for the plugin, Cordova apps cannot start the Java plugin execution. Moreover, when the mobile user denies sending information to the outside for the plugin, the proposed technique restricts the information sent to the JavaScript plugin so as not to include device resources.

4.4.2 Considering Information that the Dialog Displays to the Mobile User

Since the mobile user decides whether the Cordova app accesses device resources for the plugin, it is necessary for the mobile user to understand the plugin operation. Therefore, the proposed technique displays the plugin name and the device resources requested. In addition to the above information, when the proposed technique detects sending information, the technique displays the information that it attempts to send to the outside.

4.4.3 Avoiding Repeated Dialogs

In order to avoid repeatedly asking the mobile user for the same access, we use a plugin permission list. The plugin name is added to the permission list when the mobile user grants access to device resources for the plugin. In addition, the proposed technique confirms whether the detected plugin name is in the plugin permission list before displaying a dialog. When the plugin name exists in the plugin permission list, the proposed technique updates the plugin permission. In addition, when the detected plugin name is in the plugin permission list, the permission is granted without prompting the mobile user.

4.5 Flow of Access to Device Resources for Plugins

Figure 9 shows the flow of access to device resources for plugins using the proposed technique. Moreover, **Fig. 10** and **Fig. 11** show the flow of the proposed technique. As shown in the Fig. 10, the proposed technique controls access to the Java plugin as follows:

(1) The access control mechanism determines whether a detected plugin is in the plugin permission list.
  (A) When the plugin name is not in the plugin permission list, a dialog is shown to the mobile user.
  (B) When the plugin name is in the plugin permission list, the Cordova app starts the Java plugin execution and calls the Java method corresponding to the JavaScript API.
(2) The mobile user decides whether the Cordova app may send device resources based on the information presented in the dialog.
  (A) When the mobile user grants access to device resources, the plugin name is added to the plugin permission list.
  (B) When the mobile user denies access to device resources, the JavaScript plugin completes execution.

In addition, as shown in Fig. 11, the proposed technique controls information sent to the JavaScript plugin as follows:

(1) The access control mechanism determines whether sending device resources from a detected plugin has been granted by

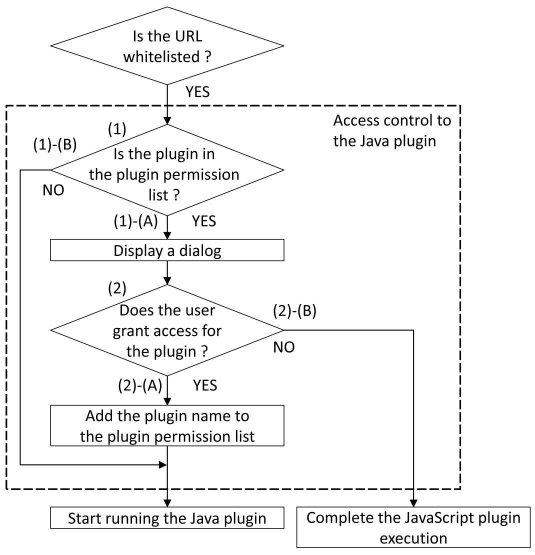

Is the URL whitelisted ?
YES
Access control to the Java plugin
(1)
(1)-(B)
Is the plugin in the plugin permission list ?
NO
(1)-(A)
YES
Display a dialog
(2)
Does the user grant access for the plugin ?
(2)-(B)
NO
(2)-(A)
YES
Add the plugin name to the plugin permission list
Start running the Java plugin
Complete the JavaScript plugin execution

Fig. 10 Flow of access control to the Java plugin.

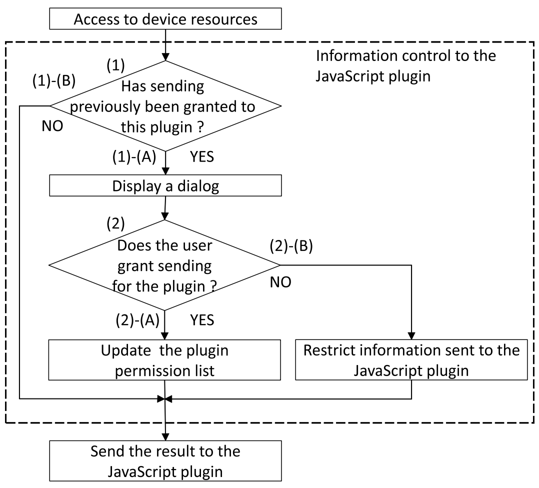

Access to device resources
Information control to the JavaScript plugin
(1)
(1)-(B)
Has sending previously been granted to this plugin ?
NO
(1)-(A)
YES
Display a dialog
(2)
Does the user grant sending for the plugin ?
(2)-(B)
NO
(2)-(A)
YES
Update the plugin permission list
Restrict information sent to the JavaScript plugin
Send the result to the JavaScript plugin

Fig. 11 Flow of information control to the JavaScript plugin.

referring the plugin permission in the plugin permission list.

(A) When sending device resources from the plugin has been granted, a dialog is shown to the mobile user.

(B) When sending device resources from the plugin has not been granted, the Java plugin sends the result to the JavaScript plugin.

(2) The mobile user decides whether the Cordova app may send the information based on the information presented in the dialog.

(A) When the mobile user grants sending the information to the JavaScript plugin, the mechanism updates the plugin permission list to record that the sending of the plugin has been granted.

(B) When the mobile user denies sending the information to the JavaScript plugin, the mechanism restricts information sent to the JavaScript plugin so as not to include device resources.

5. Implementation and Evaluation

5.1 Implementation

We implemented the proposed technique in the Cordova framework so that app developers can integrate it into existing Cordova apps more easily. The proposed technique changes the control-flow for the JavaScript plugin to access the Java plugin, restricting access to device resources.

Therefore, to implement our access control mechanism, we modified the Java implementation of the Cordova framework related to the original control-flow. In particular, we modified two Java classes (PluginManager and CallbackContext) and added five Java classes in the Cordova framework.

When app developers integrate our access control mechanism into existing Cordova apps, it is only necessary for developers to modify the above-mentioned seven Java classes in the original Cordova framework. Moreover, developers do not need to modify their app source code for introducing the proposed technique access to the Java plugin. Therefore, our access control mechanism is easy for developers to introduce to existing Cordova apps.

Table 1 Smartphone specifications.

OS	Android 6.0.1
CPU	Snapdragon 810 2.0 GHz (octa-core)
Memory	3 GB

5.2 Experimental Setup

We evaluate the proposed technique from two aspects: effectiveness in detecting *Cordova plugin attacks* and performance of the proposed technique. First, we show that the proposed technique can prevent malicious JavaScript code from exploiting the plugin API to access device resources using a sample app that we developed. Then, we consider the possibility of an *app-repackaging attack* and test applying for the proposed technique against existing Cordova apps. Finally, we evaluate the processing time of the Cordova framework using the proposed technique and the original Cordova framework, using several existing Cordova apps for our tests. We refer to the Cordova framework introduced using the proposed technique as the modified Cordova framework. **Table 1** shows the evaluation environment. We used a smartphone (Nexus 6p) for the evaluation.

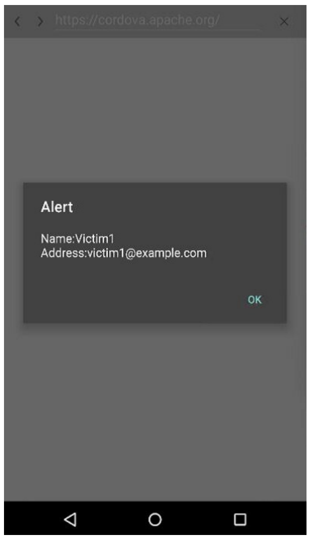
5.3 Experiment to Prevent JavaScript Code from Exploiting Plugins

We tested whether the proposed technique can prevent the JavaScript code from exploiting the plugin API to access and send device resources in a test app as shown in Section 3.2. This app displays Apache Cordova's webpage and accesses the InAppBrowser plugin and the Contacts plugin. We injected malicious JavaScript code into the test app. This code attempts to access the Contacts plugin and display the user's contacts.

Figure 12 shows a dialog displayed by the test app with the injected JavaScript code that exploits the Contacts plugin. The dialog informs the mobile user of the attempt to steal contacts before displaying Apache Cordova's webpage. Next, **Fig. 13** shows a dialog displayed by the test app built with our framework; when access to the Contacts plugin is requested, the user is asked whether to authorize the access. Therefore, Fig. 13 shows that the proposed technique can detect plugins before accessing device resources. **Figure 14** shows a dialog displayed by the test app

Table 2 Processing time of access to device resources against existing Cordova apps.

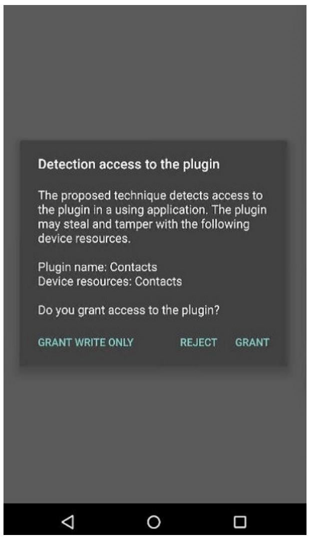
Method	Application Name				
	Aprender ingles con Wlingua	Kite Fighting	Period Calendar, Cycle Tracker	Pirate Treasures	Translator Women's Voice
(1) Original Cordova framework	1.740 ms	1.534 ms	3.426 ms	3.592 ms	3.214 ms
(2) Modified Cordova framework (first access attempt)	5.012 ms	2.812 ms	4.726 ms	4.468 ms	5.059 ms
(3) Modified Cordova framework (subsequent access attempts)	2.161 ms	2.381 ms	3.614 ms	4.130 ms	4.218 ms



Alert

Name:Victim1
Address:victim1@example.com

Fig. 12 Dialog of the injected test app in uninstrumented Cordova.



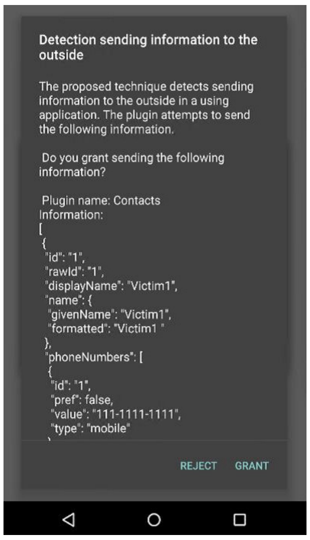
Detection access to the plugin

The proposed technique detects access to the plugin in a using application. The plugin may steal and tamper with the following device resources.

Plugin name: Contacts
Device resources: Contacts

Do you grant access to the plugin?

Fig. 13 Dialog of the test app with our defense when access to the plugin is detected.



Detection sending information to the outside

The proposed technique detects sending information to the outside in a using application. The plugin attempts to send the following information.

Do you grant sending the following information?

Plugin name: Contacts
Information:

```
[
 {
  "id": "1",
  "rawId": "1",
  "displayName": "Victim1",
  "name": {
   "givenName": "Victim1",
   "formatted": "Victim1 "
  },
  "phoneNumbers": [
   {
    "id": "1",
    "pref": false,
    "value": "111-1111-1111",
    "type": "mobile"
```

Fig. 14 Dialog of the test app with our defense when sending information to the plugin is detected.

when sending information by the Contacts plugin. The user is asked whether to authorize sending the information. In addition, we tested that the proposed technique can prevent plugins from accessing device resources when the mobile user denies access to the plugin.

Consequently, we demonstrated that the proposed technique can detect and prevent attacks that attempt to exploit Cordova's plugin interface.

5.4 Application for Existing Cordova Apps

We apply the proposed technique for existing Cordova apps. First, we chose five free Cordova apps that each have over a million downloads from Google Play. The list of Cordova apps is in **Table 2**. Next, we inject the JavaScript code into five Cordova apps by using an *app-repackaging attack*. In consequence, we could inject the JavaScript code into all Cordova apps. Therefore, it is assumed that *app-repackaging attacks* can occur realistically.

Then, we developed modified Cordova apps using the modified Cordova framework against their apps and tested whether the proposed technique can detect access to device resources for plugins. The result of applying the modified Cordova framework shows that the proposed technique did not find false positives against five Cordova apps and could detect all access to device resources for plugins against five Cordova apps. Therefore, the proposed technique can apply for existing Cordova apps.

5.5 Evaluation of Performance Overhead

To compare the performance of the original Cordova framework and the modified Cordova framework, we evaluated them against existing Cordova apps.

First, we developed modified Cordova apps using each Cordova framework against the Cordova apps shown in Section 5.4. Next, we executed the Cordova apps three times and measured the average processing time of access to device resources for plugins in the following cases.

(1) Original Cordova framework
(2) Modified Cordova framework on the first access attempt
(3) Modified Cordova framework on subsequent access attempts

Note that case (2) measures only the time of showing a dialog and the processing time of accessing device resources or sending information to the outside after the user's response. Thus, this case does not consider the time taken by the mobile user to decide whether to allow access to device resources or sending information for the plugin.

Table 2 and **Table 3** show the evaluation results. When the Cordova framework accesses device resources, Table 2 shows that the overhead on the first access is within about 1.2–3.3 ms and the overhead on subsequent access attempts is within about 0.2–1.1 ms. The maximum overhead on the first access is about 3.3 ms, which has little effect on the usability of Cordova apps. When mobile users grant access to device resources for plugins, the overhead is reduced within about 0.2–1.1 ms on future access attempts.

On the other hand, when the Cordova framework sends information, Table 3 shows that the overhead on the first sending is within about 1.6–2.5 ms and the overhead on subsequent send-

Table 3 Processing time of sending information against existing Cordova apps.

Method	Application Name				
	Aprender ingles con Wlingua	Kite Fighting	Period Calendar, Cycle Tracker	Pirate Treasures	Translator Women's Voice
(1) Original Cordova framework	0.261 ms	0.190 ms	0.108 ms	0.076 ms	0.138 ms
(2) Modified Cordova framework (first access attempt)	2.130 ms	1.778 ms	2.617 ms	2.114 ms	1.855 ms
(3) Modified Cordova framework (subsequent access attempts)	0.789 ms	0.785 ms	0.763 ms	0.613 ms	0.848 ms

ing attempts is within about 0.5–0.7 ms. The maximum overhead on the first sending is about 2.5 ms, which has little effect on the usability of Cordova apps as with the case of access to device resources. When mobile users grant sending information to the outside for plugins, the overhead is reduced within about 0.5–0.7 ms on future access attempts.

The results of both cases shows that the overhead has little effect on the usability of existing Cordova apps on the first attempt and is reduced on future access attempts. Therefore, existing Cordova apps using the modified framework remain usable.

6. Related Work

Jin et al. [1] and Georgiev et al. [4] discussed a new form of attack targeting hybrid apps. In addition, to address these attacks and improve the security of hybrid apps, NoFrak [4], Jin et al. [5], and Mohamed et al. [6] proposed fine-grained access control mechanisms for hybrid apps. Previous researches do not consider access control based on the mobile user's judgement. In contrast, MobileIFC [7] is a novel framework where the mobile user and the developer can set access permissions by specifying a resource's URL. However, mobile developers need to integrate the MobileIFC code into existing Cordova apps. Therefore, they must heavily modify their source code to introduce MobileIFC. Our proposed technique only modifies the Cordova framework. Therefore, mobile developers do not need to modify their source code to introduce the proposed technique.

On the other hand, to improve the security of the Android platform, Backes et al. [13], Nauman et al. [14], Wang et al. [15], Conti et al. [16], Bugiel et al. [17], and Yu et al. [18] proposed fine-grained access control mechanisms on Android. In the range of control objects, previous researches [13], [14], [15], [16], [17], [18] showed that control can be achieved regardless of the Android application type. The proposed technique can control when an Android application uses the Cordova framework. Therefore, previous researches have a wide range of scope compared to the proposed technique.

Next, we compare to previous researches from the viewpoint of ease of introduction. Previous researches must modify either the Android OS or the Android framework to introduce access control mechanisms proposed by previous researches because the Android OS and the Android framework do not deploy the fine-grained access control of previous researches. Therefore, when mobile users do not modify them with these defenses, they cannot deal with *Cordova plugin attacks* and other attacks. In addition, it is difficult to modify the Android OS or the Android framework of their own device. On the other hand, the proposed technique is easier for introducing users to this process of mitigating these attacks because mobile users must install only Cordova apps using our defense without modifying the Android OS or framework.

Furthermore, the proposed technique does not require configuration by users before they use it. Thus, users can immediately start using the proposed technique.

As described previously, the proposed technique can be easily introduced to users because mobile users only need to install Cordova apps using our defense mechanism. In addition, the proposed technique targets only Android applications using the Cordova framework. Therefore, compared to previous researches, our mechanism is more effective when mobile users use Android applications using the Cordova framework. Furthermore, we believe that Android applications using the Cordova framework will be more convenient and thus more popular, as mobile developers can share source code among different platforms.

7. Conclusion

In this paper, we presented a novel *app-repackaging attack* against Android Cordova apps. This attack can steal and tamper with device resources from JavaScript by exploiting Cordova's plugin interface. In addition, to mitigate against *app-repackaging attacks* and cross-site scripting attacks [1], we proposed an access control mechanism that restricts access to plugins before accessing device resources at runtime.

The proposed technique can control access to device resources for plugins and the information sent to the outside based on the user's judgement. Moreover, the proposed technique only needs to modify the Cordova framework. Therefore, in comparison with related work, it is easier to introduce our defense to existing Cordova apps. With our modified framework, mobile users can restrict access to device resources when using a compromised app. Thus, mobile users can use Cordova apps more safely. Moreover, we evaluated the effectiveness and performance of the proposed technique. The result of our testing shows that the proposed technique can prevent JavaScript by exploiting the Cordova's plugin interface from accessing device resources with little overhead. Therefore, Cordova apps are still usable with our modified framework.

In future work, we will consider an access control mechanism not heavily depending on the app user's decision to protect the sensitive information.

References

[1] Jin, X., Hu, X., Ying, K., Du, W., Yin, H. and Peri, G.N.: Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation, *Proc. 2014 ACM SIGSAC Conference on Computer and Communications Security* (*CCS '14*), pp.66–77 (2014).

[2] Apache Software Foundation: Apache Cordova, Apache Software

Foundation (online), available from ⟨https://cordova.apache.org/⟩ (accessed 2017-01-17).

[3] AppBrain: AppBrain (online), available from ⟨http://www.appbrain.com/stats/libraries/details/phonegap/phonegap-apache-cordova⟩ (accessed 2017-01-17).

[4] Georgiev, M., Jana, S. and Shmatikov, V.: Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks, *Proc. 2014 Network and Distributed System Security* (*NDSS '14*), pp.1–15 (2014).

[5] Jin, X., Wang, L., Luo, T. and Du, W.: Fine-Grained Access Control for HTML5-Based Mobile Applications in Android, *Proc. 16th Information Security Conference* (*ISC 2013*), pp.309–318 (2013).

[6] Mohamed, S. and Abeer, A.: Reducing Attack Surface on Cordova-based Hybrid Mobile Apps, *Proc. 2nd International Workshop on Mobile Development Lifecycle* (*MobileDeLi '14*), pp.1–8 (2014).

[7] Kapil, S.: Practical Context-Aware Permission Control for Hybrid Mobile Applications, *Proc. 16th International Symposium on Research in Attacks, Intrusions and Defenses* (*RAID 2013*), pp.307–327 (2013).

[8] Brucker, A.D. and Herzberg, M.: On the Static Analysis of Hybrid Mobile Apps, *Proc. 8th International Symposium on Engineering Secure Software and Systems* (*ESSoS 2016*), pp.72–88 (2016).

[9] pxb1988: dex2jar (online), available from ⟨https://github.com/pxb1988/dex2jar⟩ (accessed 2017-01-17).

[10] Emmanuel Dupuy: Java Decompiler (online), available from ⟨http://jd.benow.ca/⟩ (accessed 2017-01-17).

[11] ProGuard: ProGuard (online), available from ⟨http://proguard.sourceforge.net/⟩ (accessed 2017-01-17).

[12] Mutchler, P., Safaei, Y., Doupe, A. and Mitchell, J.: Target Fragmentation in Android Apps, *Proc. IEEE Security Privacy Mobile Security Technologies Workshop* (*MoST*) (2016).

[13] Backes, M., Bugiel, S., Gerling, S. and von Styp-Rekowsky, P.: Android Security Framework: Extensible Multi-Layered Access Control on Android, *Proc. 30th Annual Computer Security Applications Conference* (*ACSAC '14*), pp.46–55 (2014).

[14] Nauman, M., Khan, S. and Zhang, X.: Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints, *Proc. 5th ACM Symposium on Information, Computer and Communications Security* (*ASIACCS '10*), pp.328–332 (2010).

[15] Wang, Y., Hariharan, S., Zhao, C., Liu, J. and Du, W.: Compac: Enforce Component-Level Access Control in Android, *Proc. 4th ACM Conference on Data and Application Security and Privacy* (*CODASPY '14*), pp.25–36 (2014).

[16] Conti, M., Nguyen, V.T.N. and Crispo, B.: CRePE: Context-Related Policy Enforcement for Android, *Proc. 13th International Conference on Information Security*, pp.331–345 (2010).

[17] Bugiel, S., Heuser, S. and Sadeghi, A.-R.: Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies, *Proc. 22nd USENIX Conference on Security*, pp.131–146 (2013).

[18] Yu, J. and Yamauchi, T.: Access Control to Prevent Malicious JavaScript Code Exploiting Vulnerabilities of WebView in Android OS, *IEICE Trans. Inf. Syst.*, Vol.E98-D, No.4, pp.807–811 (2015).

Editor's Recommendation

The aim of the paper is clearly stated and the problems are carefully addressed. The proposed method exploits a permission-based access control scheme, which might be rather a well-known approach, however, the evaluation results of the proposed method implemented on a real system should be valuable for readers and thus is selected as a recommended paper.

(Program chair of Computer Security Symposium 2016, Masayuki Terada)

Naoki Kudo received his B.E. and M.E. degrees from Okayama University, Japan in 2015 and 2017 respectively. His research interests include computer security and Android platforms.

Toshihiro Yamauchi received his B.E., M.E. and Ph.D. degrees in computer science from Kyushu University, Japan in 1998, 2000 and 2002, respectively. In 2001 he was a Research Fellow of the Japan Society for the Promotion of Science. In 2002 he became a Research Associate in the Faculty of Information Science and Electrical Engineering at Kyushu University. He has been serving as an Associate Professor of the Graduate School of Natural Science and Technology at Okayama University since 2005. His research interests include operating systems and computer security. He is a member of IPSJ, IEICE, ACM, USENIX, and IEEE.

Thomas H. Austin earned his Ph.D. in computer science from UC Santa Cruz. He is currently an Assistant Professor at San Jose State University, and has previously worked with Mozilla's research group, ESIEA Ouest's Cryptology and Operational Virology lab, and CloudFlare, Inc. His research interests include malware analysis and programming language design.