Access Control for Plugins in Cordova-based Hybrid Applications

Naoki Kudo and Toshihiro Yamauchi
Graduate School of Natural Science and Technology,
Okayama University, Okayama, Japan
Email: yamauchi@cs.okayama-u.ac.jp

Thomas H. Austin
San Jose State University, San Jose, USA
Email: thomas.austin@sjsu.edu

***Abstract*—Hybrid application frameworks such as Cordova allow mobile application (app) developers to create platform-independent apps. The code is written in JavaScript, with special APIs to access device resources in a platform-agnostic way. 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 tamper with device resources. We further demonstrate a defense against this attack 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 *app-repackaging attacks*.**

I. Introduction

Hybrid application (app) frameworks are more and more popular in developing platform-independent apps. Unlike conventional mobile apps, hybrid apps is 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 [1]. 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 tamper with device resources by exploiting Cordova's plugin interface. In addition to these attacks, we address cross-site scripting attacks against hybrid apps [2]. These attacks need to use plugins to access device resources. To address these 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 [3], Jin et al. [4], and Mohamed et al. [5]. None of the previous research considered access control based on a mobile user's judgement. In contrast, MobileIFC [6] 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, our mechanism 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 *app-repackaging attacks* and cross-site scripting attacks. Our mechanism is easy for app developers to introduce to existing Cordova apps since they do not need to modify the app's source code.

II. Cordova Apps

A. Structure of Cordova Apps

1) Structure: Fig. 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 the 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 by using the bridge.
(2) The Cordova app accesses device resources from the Java plugin.

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

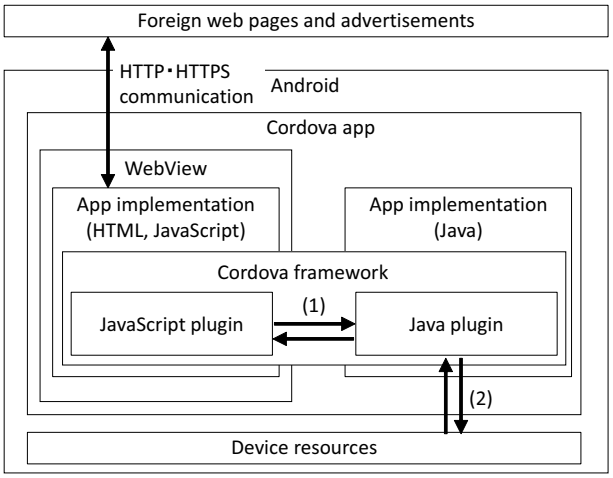

Fig. 1. Structure of Cordova apps on Android.

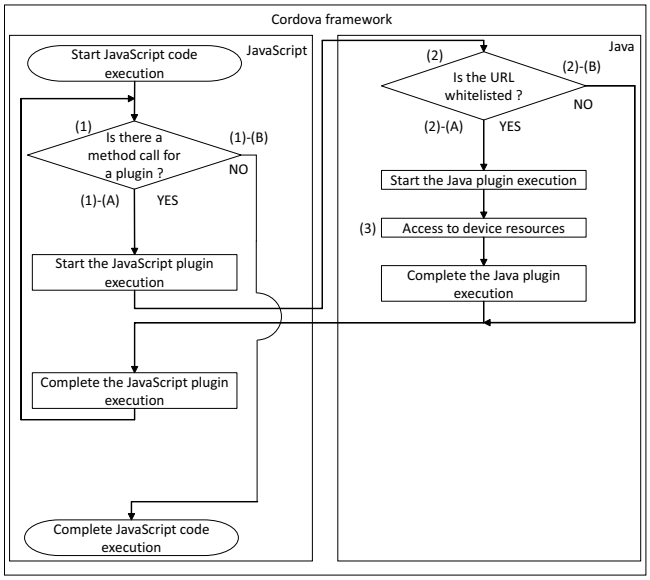

Fig. 2. Flow of access to device resources for plugins in Cordova framework.

Java methods, which can access device resources. Cordova apps can access device resources through JavaScript code by using JavaScript APIs.

B. Flow of Access to Device Resources for Plugins

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

(1) Cordova determines whether JavaScript code calls a JavaScript method for the plugin.
   (A) When JavaScript code calls the JavaScript method for the plugin, the Cordova app starts the JavaScript plugin execution and calls the JavaScript API.
   (B) When 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 and then the Java plugin and the JavaScript plugin completes execution.

C. 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 tamper with device resources through JavaScript.

III. Cordova Plugin Attacks

A. 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 tamper with device resources through JavaScript as mentioned in Section II-C. In addition, Jin et al. [2], Mohamed et al. [5], and Brucker et al. [7] show that malicious attackers can 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:

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

We refer to both of these attacks as *Cordova plugin attacks*, since they leverage 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.

B. App-Repackaging Attack

1) Structure of Apk Files for Android Cordova Apps: Before explaining how to inject JavaScript code by repackaging Cordova apps, we show the architecture of apk files on Android Cordova apps in Fig. 3. The app source code of HTML and JavaScript are stored in /assets/www/*.

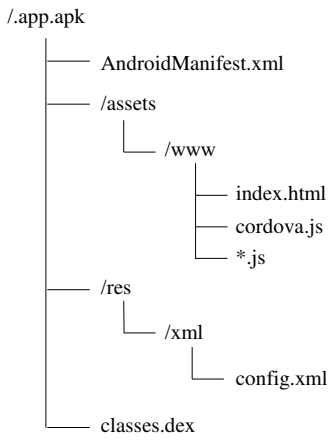

Fig. 3. Structure of apk files for Android Cordova apps.

2) How to Inject JavaScript by Repackaging Cordova Apps: The process of injecting JavaScript code by repackaging Cordova apps takes place as follows:

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

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

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 classes.dex by using reverse engineering tools such as dex2jar [8] and Java Decompiler [9]. Moreover, they disassemble classes.dex 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 [10] to obfuscate 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 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 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*.

C. Cross-Site Scripting Attack on Cordova Apps

In this section, we explain cross-site scripting attacks demonstrated in Jin et al. [2]. In a typical cross-site scripting attack, attackers inject JavaScript code into the data field (such as in a form). Since we 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 JavaScript code compared to web applications. In one example, Jin et al. inject HTML tags and JavaScript code into an existing hybrid app by using QR code and steal a device's geolocation.

D. Discussion

In this section, we consider whether 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. [11], 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*.

IV. Access Control for Plugins

A. Concept of Proposed Technique

To mitigate *Cordova plugin attacks* as described in Section III-A, we propose an access control mechanism that restricts

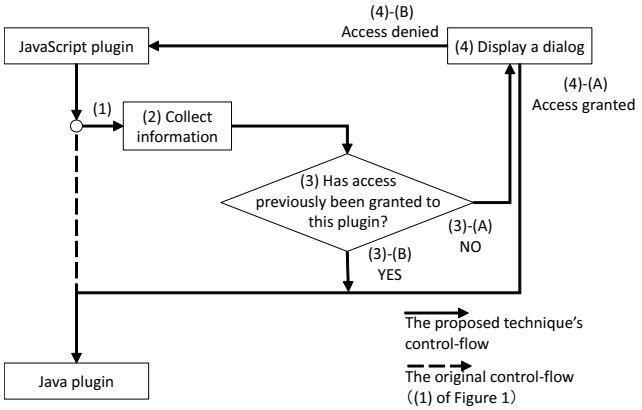

Fig. 4. Overview of the proposed technique.

access to plugins before accessing device resources at runtime. The purpose of the proposed technique is to prevent malicious JavaScript code from exploiting Cordova's plugins to access 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 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.

B. Design

Fig. 4 shows an overview of the proposed technique, which controls access to device resources from 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.

C. 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 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 device 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 are inconvenient the mobile user.

D. Our Solution

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 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.

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.

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 detected plugin name is in the plugin permission list, the permission is granted without prompting the mobile user.

E. Flow of Access to Device Resources for Plugins

Fig. 5 shows the flow of access to device resources for plugins using the proposed technique. The flow of the proposed technique's access control is as follows:

(1) First, 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 control is moved to (3) and the Cordova app starts the Java plugin execution.

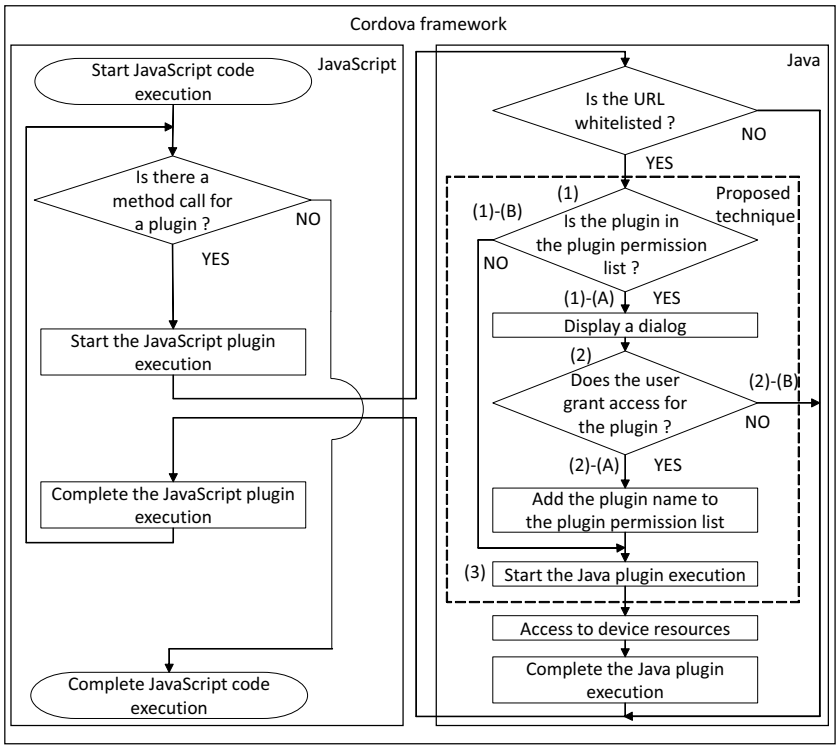

Fig. 5. Flow of access to device resources for plugins using the proposed technique in Cordova framework.

(2) The mobile user decides whether the Cordova app may access 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.

(3) The Cordova app starts the Java plugin execution and calls the Java method corresponding to the JavaScript API.

V. Implementation and Evaluation

A. 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 one Java class (PluginManager) 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 six 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 I
SMARTPHONE SPECIFICATIONS.

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

B. Experimental Setup

We evaluate the proposed technique on two aspects: effectiveness in detecting against *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 I shows the evaluation environment. We used a smartphone (Nexus 6p) for the evaluation.

TABLE II
PROCESSING TIMES 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

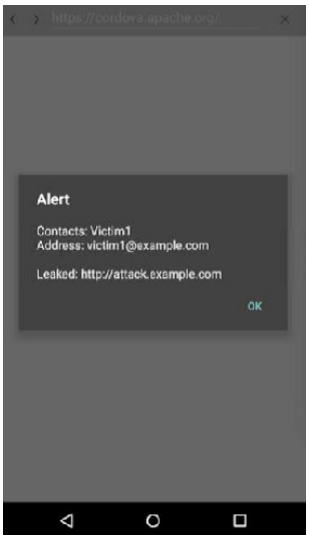

Fig. 6. Dialog of the injected test app in uninstrumented Cordova.

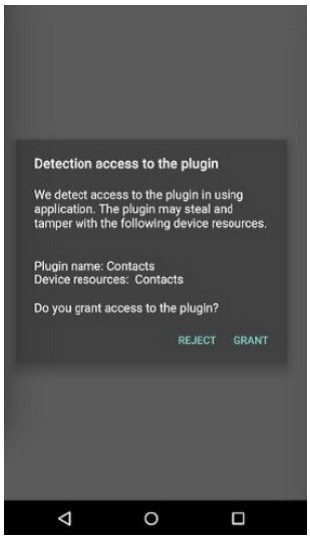

Fig. 7. Dialog of the test app with our defense.

C. Experiment to Prevent JavaScript Code from Exploiting Plugins

We tested whether the proposed technique can prevent JavaScript code from exploiting the plugin API to access device resources in a test app that we developed. 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.

Fig. 6 shows a dialog displayed by the test app with injected JavaScript code that exploits the Contacts plugin. The dialog informs the mobile user of contacts leakage before displaying Apache Cordova's webpage. Next, Fig. 7 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. 7 shows that the proposed technique can detect plugins before accessing device resources. 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.

D. 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 II. Next, we inject JavaScript code into five Cordova apps by using an *app-repackaging attack*. In consequence, we could inject 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 test these 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 can for the five Cordova apps and did detect all access to device resources for plugins. Therefore, the proposed technique can apply for existing Cordova apps.

E. 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 V-D. 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 accessing device resources 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 for the plugin.

Table II shows the evaluation result. Table II 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 have little effect on the usability of Cordova apps. Moreover, 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. Therefore, existing Cordova apps using our framework remain usable.

VI. Related Work

Jin et al. [2] and Georgiev et al. [3] discussed a new form of attack targeting hybrid apps. In addition, to address these attacks and improve security of hybrid apps, NoFrak [3], Jin et al. [4], and Mohamed et al. [5] proposed fine-grained access control mechanisms for hybrid apps. Previous research does not consider access control based on the mobile user's judgement. In contrast, MobileIFC [6] 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 security of Android, Backes et al. [12], Nauman et al. [13], Wang et al. [14], Conti et al. [15], Bugiel et al. [16], and Yu et al. [17] proposed fine-grained access control mechanisms on Android. Previous research modifies the Android OS and the Android framework, requiring mobile users to replace them with these defenses. Therefore, when mobile users do not replace them with these defenses, they cannot address *Cordova plugin attacks*. The proposed technique is easier to introduce for mitigating against these attacks than previous research because mobile users install Cordova apps using our defense.

VII. Conclusion

In this paper, we presented a novel *app-repackaging attack* against Android Cordova apps. This attack can 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 [2], we proposed an access control mechanism that restricts access to plugins before accessing device resources at runtime.

The proposed technique can detect access to device resources for plugins and control access 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 Cordova's plugin interface from accessing device resources and Cordova apps are still usable with our modified framework.

In future work, we will consider access control targets of plugins and reconsider the information that a dialog shows to the mobile user for improving the proposed technique's convenience.

References

[1] Apache Cordova. [Online]. Available: https://cordova.apache.org/

[2] 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 the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, 2014, pp. 66–77.

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

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

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

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

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

[8] dex2jar. [Online]. Available: https://github.com/pxb1988/dex2jar

[9] Java Decompiler. [Online]. Available: http://jd.benow.ca/

[10] ProGuard. [Online]. Available: http://proguard.sourceforge.net/

[11] P. Mutchler, Y. Safaei, A. Doupe, and J. Mitchell, "Target Fragmentation in Android Apps," in *Proceedings of the IEEE Security Privacy Mobile Security Technologies Workshop (MoST)*, 2016.

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

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

[14] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce Component-Level Access Control in Android," in *Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY '14)*, 2014, pp. 25–36.

[15] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-Related Policy Enforcement for Android," in *Proceedings of the 13th international conference on Information security*, 2010, pp. 331–345.

[16] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," in *Proceedings of the 22nd USENIX conference on Security*, 2013, pp. 131–146.

[17] J. Yu and T. Yamauchi, "Access Control to Prevent Malicious JavaScript Code Exploiting Vulnerabilities of WebView in Android OS," *IEICE Transactions on Information and Systems*, vol. E98-D, no. 4, pp. 807–811, 2015.