Access Control to Prevent Attacks Exploiting Vulnerabilities of WebView in Android OS

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

***Abstract*—Android applications that using WebView can load and display web pages. Furthermore, by using the APIs provided in WebView, Android applications can interact with web pages. The interaction allows JavaScript code within the web pages to access resources on the Android device by using the Java object, which is registered into WebView. If this WebView feature were exploited by an attacker, JavaScript code could be used to launch attacks, such as stealing from or tampering personal information in the device. To address these threats, we propose a method that performs access control on the security-sensitive APIs at the Java object level. The proposed method uses static analysis to identify these security-sensitive APIs, detects threats at runtime, and notifies the user if threats are detected, thereby preventing attacks from web pages.**

I. Introduction

In the last several years, the Android [1], a mobile platform proposed and developed by Google, has become more popular. To support the development of Android applications (hereafter, Android apps), Android provides rich libraries, such as OpenGL, SQLite, WebKit, etc. In this paper, we focus on WebKit. WebView [2], provided by WebKit, could be used to implement a simple browser function in an Android app, so that users can load web pages in the Android app directly without using a browser. In addition to displaying web pages, WebView allows JavaScript within web pages to invoke methods defined in the Android apps. However, if the rich features of WebView are not used properly, devices could become vulnerable to malicious attacks [3], such as those that steal personal information or tamper with data on the Android device.

Reference [4] has reported attacks using the vulnerabilities of WebView in Android, and these attacks fall into two types: attacks from web pages to the Android OS and attacks from Android apps to web pages. Reference [5] was first to recognize the threat of JavaScript code that abuse Android permissions, and the authors proposed a static analysis method to estimate the potential threat of Android apps that use WebView. However, no effective countermeasures were discussed in existing works.

In this paper, we study a case of an attack from web pages, and we address the cause of the threats, which is the access to security-sensitive APIs from the JavaScript code within the web pages. We propose access control on the security-sensitive APIs at the Java object level. The Java object is an interface to the web pages loaded in WebView. By using the Java object, JavaScript code in the web pages can access the resources on the Android device. To detect potential attacks that use the Java object, we performed static analysis to determine the security-sensitive APIs that could be invoked by the Java object, and we ran a threat-detection process each time the Java object is registered into WebView. If a threat is found, the user is warned, and the user decides whether to allow the registration of the Java object or to disable it to prevent attacks from web pages.

The contributions of this work are as follows:

- Detect all threats that come from web pages. Because all Java objects are registered by using the addJavascriptInterface API, we hooked the addJavascriptInterface API, so that every Java object that is intended to be registered into WebView can be inspected.
- Support the user in making a decision. By displaying the URL of the web page to be loaded in WebView, we can enhance the awareness of the threat that could come from web pages. By referring to information about the URL and the security-sensitive APIs, the user could evaluate the threat. If the user is not certain of the safety of the Java object, the user could use the default browser to load the web page, instead of loading it in WebView, by clicking the URL displayed in the warning information.
- Prevent the threats without stopping the Android app. Because the proposed method performs access control at the Java object level, the user disables only the Java object. The Android app could continue running, and the user could browse the web page in WebView safely.

II. Security Components

A. Dalvik Virtual Machine

Dalvik is the virtual machine that runs Android apps on the Android OS. In addition to hiding the specification differences among devices, the Dalvik virtual machine performs as a sandbox to isolate Android apps from each other. Because each Android app has its own UID and runs on the Dalvik virtual machine independently, the security is enhanced.

Dalvik executable files are formatted as dex (Dalvik Executable) files. An Android app written in Java is compiled and converted to a dex file, which contains all the source code of the Android app. The dex file is also used in our static analysis.

B. Permission

The permission mechanism performs access control on the resources that Android apps can access. If permissions are requested by an Android app, the user is prompted at installation time. However, the installation continues, only if the user grants all requested permissions. The user cannot grant only some of the permissions. In addition, the permissions cannot be changed after the Android app is installed. Because the current permission mechanism is coarse-grained, the Android app can trick the user to grant more permissions and be over-privileged.

III. WEBVIEW

A. What is WebView

WebView is a component provided by a browser engine named WebKit. WebView provides basic browser functionality to load and display Web pages within Android apps without switching to the default browser. More importantly, the Android app can interact with JavaScript code embedded in web pages by using the APIs provided in WebView. Therefore, developers can develop an Android app with rich features by using WebView. In this section, we introduce the most important APIs used in the interaction between the Android app and web pages.

1) setJavaScriptEnabled API: This API enables web pages to use JavaScript, which plays an important role in the interaction. Web pages using JavaScript might not be successfully displayed, if the JavaScript function is not enabled.

2) addJavascriptInterface API: This mechanism allows web pages to execute a method defined in the Android. The addJavascriptInterface API registers the Java object into WebView, so that the JavaScript code within web pages could use the registered Java object to run methods defined in the Java class.

3) loadUrl API: This API loads a specific web page. The following example shows how to load the home page of Google.

```
WebView webview = new WebView(this);
webview.loadUrl("http://www.google.com/");
```

JavaScript code can also be injected into web pages by using loadUrl API. The following is an example that inserts the alert “Hello WebView” in the Google home page.

```
webview.loadUrl("http://www.google.com/
        javascript:alert("Hello WebView")");
```

B. Problems with WebView and Attack Models

WebView enables Android app to function as a simple browser. However, compared with the general browsers, WebView is not secure enough. Fig.1 shows an overview of an Android app that uses WebView. Path A and Path B are the two main interactions between the Android app and web pages. Path A shows that web pages loaded in WebView can invoke methods defined in the Android by using registered Java objects. Path B shows that by using the loadUrl API, an Android app can invoke JavaScript code within web pages or insert JavaScript code into web pages. Attackers could exploit these rich features of WebView.

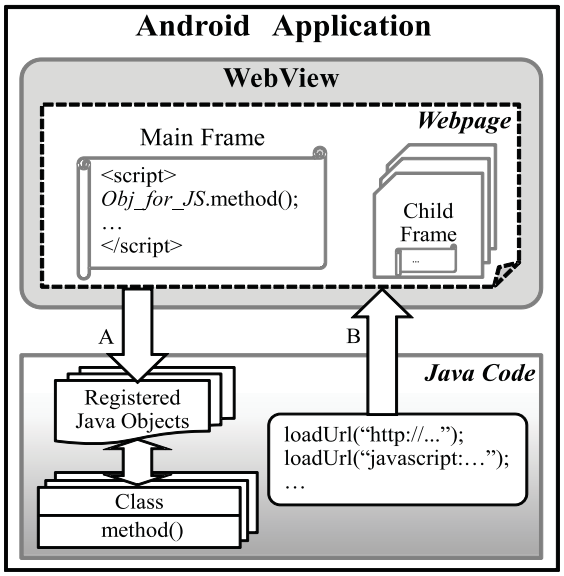

Fig. 1. Overview of Android application using WebView

Two attack models, which also have been discussed in [4], should be considered. One model involves attacks from web pages. After the Java object is registered into WebView, all web pages loaded in WebView can use the registered Java object, regardless of the origin of the web pages. If WebView loads the malicious web pages, the JavaScript code in those malicious web pages could launch an attack, such as stealing or tampering with the personal information in the Android device. Another model involves attacks from Android apps. Malicious Android apps can inject malicious JavaScript code into web pages to perform attacks. In this paper, we propose a method to address the attacks from web pages.

C. Tutorial on Attack Case

1) Case Study: Many attack examples that exploit WebView were documented in [6]. In this paper, we show an example that steals the telephone number of an Android device using JavaScript code embedded in web pages. Because Android apps need permission to access the resources and data, in this example, we assume that the Android app has received the INTERNET permission to open network sockets and the READ_PHONE_STATE permission to access the phone state.

In order to illustrate the interactions between the Android app and the web pages, we created implementations both in the Android and in the web page. Fig.2 is the example code for Android, and Fig.3 is the example code for the web page.

The first four lines of Fig.2 are necessary to invoke methods in an Android device from a JavaScript code. The first four lines do the following:

- Create an instance of the WebView
- Enable JavaScript
- Register the Java object in WebView
- Specify the URL of the web page to load

If any one of the four necessary elements is absent, the JavaScript code could not pose a risk, because it would not be able to invoke methods in the Android device.

```
1:  WebView webview = new WebView(this);
2:  webview.getSettings().setJavaScriptEnabled(true);
3:  webview.addJavascriptInterface(new JavaObject(), "Obj_for_JS");
4:  webview.loadUrl("http://target.com");
…
5: public class JavaObject () {
6:      public void method_1() {
7:          //do something
8:      }
9:      public boolean method_2(String data) {
10:         //do something
11:     }
…
12:     public string method_n() {
13:         //invoke getLine1Number() API to get telephone number
14:         phoneNum = telephonyManager.getLine1Number();
            …
15:     }
16: }
```

Fig. 2. Example code for Android

```
1: <script>
2:   var secData;
3:   secData = Obj_for_JS.method_n();
     …
4: </script>
```

Fig. 3. Example code for the web page

At Line 3 of Fig.2, a Java object named Obj_for_JS, which is an instance of the JavaObject class, is registered through the addJavascriptInterface API. At Line 3 of Fig.3, the JavaScript code uses Obj_for_JS to invoke the Android app's method_n(), which invokes the getLine1Number API of the TelephonyManager class. In this way, the JavaScript code within a web page could steal the telephone number of the Android device.

2) Analysis: As we can learn from the attack case that the Java object has the same privileges as the Java class and can invoke the methods defined in the Java class. The key point is to understand what operations the method can do. If the method in the Java class can execute the security-sensitive APIs, which could access personal information or alter data stored in the Android device, the Java object associated with this Java class could pose a threat. Therefore, it is necessary to determine whether the security-sensitive APIs are used and to apply access control on them.

IV. Proposed Method

A. Purpose

The purpose of the proposed method is to prevent malicious JavaScript code from accessing security-sensitive APIs through the Java object. The user is prompted to grant permissions requested by the Android app at installation time. However, the user is not aware that these permissions could be used by malicious JavaScript code in a web page to invoke the security-sensitive APIs through the Java object to attack. For this reason, access control must be applied on the security-sensitive APIs.

B. Concept of Proposed Method

We know that the threat from JavaScript code stems from the use of security-sensitive APIs. However, as described in the previous chapter, the Java object is one of the necessary elements that could pose a threat, and controlling the Java object can prevent attacks from JavaScript code. Therefore, we applied access control on the security-sensitive APIs at Java object level. If the security-sensitive APIs are detected in methods that the Java object could execute, we control this Java object. The following are the requirements for achieving this purpose:

a) Be able to identify whether a specific Java object needs to be controlled.

b) Be able to manage the Java object that needs to be controlled.

To achieve these requirements, the following three challenges must be met:

a) Clarify what APIs can be executed by the Java object.

b) Address the security-sensitive APIs at Java object level.

c) Prompt the user to manage the potential threat that has been addressed.

To satisfy Requirement a), Challenges a) and b) need to be resolved. To satisfy Requirement b), Challenge c) needs to be resolved.

C. Solution

1) Solution for Challenge a): To determine what APIs could be executed by Java objects, we need to refer to the source code of the associated Java class. However, the Android app is compiled and packaged in the form of apk files, and we could not refer directly to the source code of the Java class. Therefore, we use static analysis to determine what APIs are used in the Android app.

Dedexer [7] and dex2jar [8] are commonly used analysis tools. Dedexer can convert the dex file into assembly code, and dex2jar can convert the dex file into a jar file that contains the Java class file. By using the JD-GUI [9], the Java class file can be converted back to the Java code. However, these tools are used in Windows and are not available in the Android OS. Therefore, we use dexdump, which is a disassembly tool for Android, to convert the dex file into assembly code. By analyzing the assembly code, we can determine what APIs were used. In addition, to make the assembly code simple and easy to be analyzed, we removed unnecessary features of dexdump to make it lighter, thereby reducing the overhead of the static analysis.

The time when the static analysis is performed also has an effect on the performance of the proposed method. One option is during installation of the Android app, and another is when the addJavascriptInterface API is invoked. If we perform static analysis during installation, the time spent on static analysis could be perceived as part of the installation time, so the user does not feel inconvenienced. However, we cannot determine

TABLE I. SECURITY-SENSITIVE APIS

getCellLocation	getAccounts
getDeviceId	getAuthToken
getNetworkOperator	getPassword
getPhoneType	getUserData
getSubscriberId	peekAuthToken
getLine1Number	removeAccount
getSimSerialNumber	setPassword
getVoiceMailAlphaTag	getName
getVoiceMailNumber	getProfileConnectionState
sendDataMessage	getProfileProxy
sendMultipartTextMessage	getParams
sendTextMessage	getUngzippedContent
getAllProviders	getCertificate
getBestProvider	clearHistory
getGpsStatus	clearSearches
getLastKnownLocation	getAllBookmarks
clearPassword	getAllVisitedUrls
editProperties	

which application needs to be analyzed at installation time; therefore, static analysis would need to be performed on every Android app. If we perform static analysis when the addJavascriptInterface API is invoked, we are certain that the static analysis is performed only on applications that use the addJavascriptInterface API. However, the user would need to wait until the static analysis is completed. User experience is important for mobile applications; therefore, the proposed method performs the static analysis during installation.

2) Solution for Challenge b): By analyzing the assembly code generated by dexdump, we can determine which Java class includes the security-sensitive APIs that need to be controlled. In our work, we investigate on API Level 15, and we define security-sensitive APIs as the APIs that communicates with outside or that deal with personal information. TABLEI shows the list of APIs that we defined to be security-sensitive.

3) Solution for Challenge c): In Android OS, no threat detection has been done during registration of the Java object. In our method, we hook the addJavascriptInterface API at runtime to detect whether a threat exists in the Java object. Threat detection is based on the result of the static analysis. If the Java object is determined to invoke the security-sensitive APIs, the user will be warned and prompted to decide whether to grant permission based on the information presented. By disabling the dangerous Java objects, the user can prevent attacks from web pages.

D. Design

1) Overview of Proposed Method: Fig.4 is an overview of the proposed method, which mainly consists of three components: the Static Analysis Unit, the Threat Detection Unit, and the Alarm application. As shown in Fig.4, our method controls the Java object at the framework layer. By intercepting the call to the addJavascriptInterface API, the threat that exists in the registered Java object is detected and the user is informed. Then, the user can decide whether to disable the Java object. In addition, the Threat Detection Unit detects whether a potential threat exists in the Java object based on the API_Class Matching List, which is generated by the

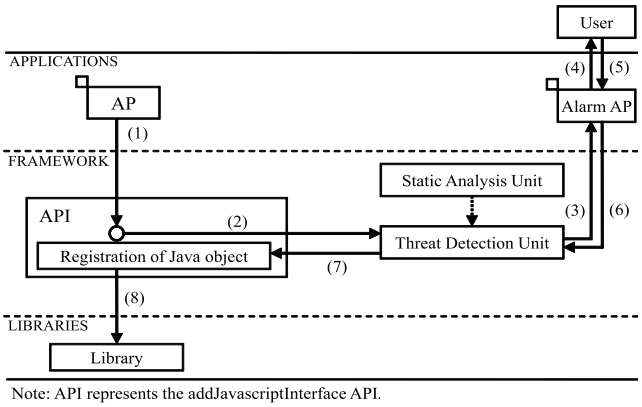

Fig. 4. Overview of proposed method

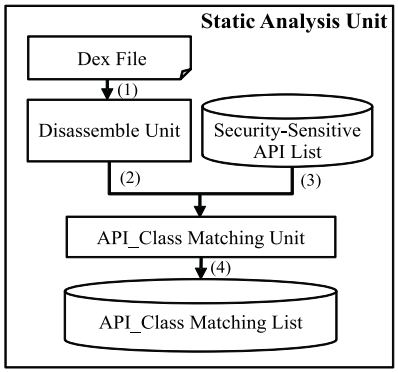

Fig. 5. Processing flow of static analysis unit

Static Analysis Unit to show the association between the Java class and the security-sensitive APIs. The dotted arrow from the Static Analysis Unit to the Threat Detection Unit represents a reference to the API_Class Matching List. The following process describes the flow in the proposed method:

(1) The Android app calls the addJavascriptInterface API.
(2) The proposed method intercepts the call to the addJavascriptInterface API and passes the information about the Java object to the Threat Detection Unit.
(3) If a threat is detected by the Threat Detection Unit, the proposed method calls the Alarm application; otherwise, it proceeds to Step (7).
(4) The proposed method warns the user of the threat.
(5) The user replies to the Alarm application to decide whether to disable the Java object.
(6) The Alarm application forwards the user's decision to the Threat Detection Unit.
(7) The Threat Detection Unit forwards the decision to the addJavascriptInterface API.
(8) If the user granted the use of the Java object, the Java object is registered into WebView as usual. Otherwise, the registration is aborted.

2) Static Analysis Unit: Fig.5 shows the processing flow of the Static Analysis Unit. The Static Analysis Unit consists of the Disassemble Unit and the API_Class Matching Unit. The following process describes the flow in the Static Analysis Unit:

(1) The Static Analysis Unit gets the dex file from the apk file of the Android app.
(2) The Disassemble Unit converts the dex format to assembly code by using dexdump.
(3) The API_Class Matching Unit compares the assembly code with the Security-Sensitive API List, which was defined previously.
(4) If the assembly code contains APIs that are included in the Security-Sensitive API List, the Static Analysis Unit stores the API information and the associated Java class information in the API_Class Matching List.

The static analysis must be done once on each Android app and again, if the Android app is updated.

3) Threat Detection Unit: When the Java object information is passed to the Threat Detection Unit, the detection is performed based on the API_Class Matching List that was generated by the Static Analysis Unit. If a threat is detected, the Alarm application would be called. The Threat Detection Unit also passes the decision made by the user to the addJavascriptInterface API.

4) Alarm AP: If a threat is detected, the Alarm application warns the user and prompts the user to make a decision whether to grant permission to the Java object. In order to support the user in making a decision, the following information is displayed:

- Name of Android app
- URL that WebView is supposed to load
- Name of Java object that was determined to be a threat
- Security-sensitive API associated with the Java object

The user can press either the "Enable" or "Disable" button located below the warning information to indicate whether to allow the use of the Java object. In addition, by clicking the URL displayed, the default browser will be invoked to load the web page. As sandbox protection is implemented in general browser, the Java object cannot be used to interact with the Android app. Therefore, if the user is not certain of the safety of the Java object, the user can use the default browser to load the web page, instead of loading it in WebView.

V. Evaluation

A. Experiment to test the operation of proposed method

We implemented the proposed method on Android 4.0.3 and experimented with the operation of the proposed method. We used an Android app named HelloWebView, which has the functionality to obtain the phone number and the device ID using JavaScript code embedded in the web page and using the Java object named Obj_for_JS. The warning information is shown in Fig.6. By choosing "Disable", the user denies access to the getLine1Number API and the getDeviceID API, which are the security-sensitive APIs that detected by the proposed method.

B. Effectiveness of modified dexdump

In the proposed method, we modified the dexdump to make the assembly code simple and easier to analyze. We compared the unmodified and the modified dexdump. The comparison

Fig. 6. Example of warning information

TABLE II. COMPARISON BETWEEN UNMODIFIED DEXDUMP AND MODIFIED DEXDUMP

Android App	A	B	C	D
The Weather Channel	4,944 KB	74,630 KB	31,193 KB	58.20%
Vyclone-Film together	4,659 KB	71,885 KB	27,494 KB	61.75%
Contacts+	3,131 KB	53,704 KB	19,633 KB	63.44%
Instructables	2,628 KB	41,272 KB	16,321 KB	60.46%
Sports Republic	3,531 KB	48,396 KB	22,865 KB	52.75%
Appy Gamer	2,496 KB	35,776 KB	13,014 KB	63.62%

Note: Column A shows the size of the dex file. Column B shows the size of the assembly code generated by the unmodified dexdump. Column C shows the size of the assembly code generated by the modified dexdump. Column D shows the rate of reduction.

details are shown in TABLEII. We used six free Android apps that downloaded from the "Recommended Apps This Week" on Google Play on June 11, 2013. The dex file of each Android app was disassembled by the unmodified dexdump and by the modified dexdump. The size of the assembly code generated by the modified dexdump was reduced by about 60%, compared to the one generated by the unmodified dexdump.

C. Overhead of the Static Analysis Unit

We measured the processing time of the static analysis. The environment we used are shown in TABLEIII and TABLEIV. We ran the guest OS using VMware Player 4.0.4 on the host OS and then ran Android 4.0.3, which the proposed method has been implemented, on the guest OS. We tested two representative Android apps using WebView: HelloWebView, which is the smallest Android app that uses WebView, and LivingSocial, which has been introduced in [4] is the size of a typical Android app that uses WebView. The static analysis was performed 5 times on each Android app, and we averaged

TABLE III. ENVIRONMENT OF THE HOST OS

OS	Windows 7 Home Premium
CPU	Intel(R) Core(TM) i7-3517U 1.90GHz
Memory	8 GB
Virtualization Software	VMware Player 4.0.4

TABLE IV. ENVIRONMENT OF THE GUEST OS

Distribution	Ubuntu 10.04 LTS
Kernel	Linux 2.6.32-44-generic
Number of virtual CPU	2
Memory	4 GB

TABLE V. PROCESSING TIMES OF STATIC ANALYSIS

Android App	Size of Dex File	Average of Processing Time
HelloWebView	5.6 KB	182 ms
LivingSocial	781.8 KB	12,324 ms

the processing times. As shown in TABLEV, the processing time of HelloWebView is 182 ms, which is very short. On the other hand, the processing time of LivingSocial is about 12 s, which needs to be reduced. However, because the static analysis is performed only once at installation, 12 s may not be a serious inconvenience to the user.

VI. Related Work

Nowadays, JavaScript has been widely used and many works have been done to enhance the security in web browsers. Reference [10] identified the fundamental lack of fine-grained JavaScript access control mechanisms in modern web browsers and proposed a method that enables fine-grained access control in JavaScript contexts. Reference [11] presented a client-side advice implementation called CONSCRIPT, which allows the hosting page to express fine-grained application-specific security policies at runtime. On the other hand, the researches on Android security are also booming. Reference [12] presented a methodology for the empirical analysis of permission-based security models and made a discussion of improvement for the coarse-grained permission model. Reference [13] adopted bytecode rewriting to implement fine-grained access control at the API level. Reference [14] proposed DroidTrack, a method for tracking the diffusion of personal information and preventing its leakage on Android device.

In this paper, we focus on WebView and make an implementation in Android OS to protect the Android device from the malicious JavaScript. Vulnerabilities caused by the use of WebView have attracted the attention of the research community [3], [6]. Reference [4] reported that WebView is used in 86% of the top 20 most downloaded Android apps in 10 different categories. Further, two attack models (attacks from malicious web pages and attacks from malicious Android apps) were discussed.

Reference [5] proposed a static analysis method, which is similar to the one we used in our method, to estimate the threat while using WebView. The static analysis technique detects threat based on a malignant API list which was previously defined. However, the threat is evaluated at the Android app level, and the user is only notified whether the Android app is dangerous or not. Therefore, the user can do nothing, except to keep the dangerous Android app unused. In addition, there is not much information prompted to the user, so that the Android app would be estimated as a dangerous one even if it is benign.

The method proposed in this paper addresses the potential threats at the Java object level and provides the information to the user. By disabling potentially malicious Java objects, the user can prevent attacks that come from web pages. The proposed method can prevent the threat to Android devices, even if web pages with malicious JavaScript are loaded in WebView.

VII. Conclusion

In this paper, we described the attacks to Android devices from web pages caused by exploiting the vulnerabilities of WebView. To resolve these attacks, we applied access control on the security-sensitive APIs at the Java object level. The threat detection is performed when the addJavascriptInterface API is invoked to register the Java object into WebView, and the user is notified if a threat is detected. By disabling the malicious Java object, attacks from web pages could be prevented. In future work, we will reduce the overhead of the proposed method.

References

[1] Android, the world's most popular mobile platform - Android Developers. [Online]. Available: http://developer.android.com/about/index.html

[2] WebView - Android Developers. [Online]. Available: http://developer.android.com/reference/android/webkit/WebView.html

[3] (2012, Jun.) Dangers lurking in the implementation of the browser features to smartphone app - issue of WebView class. (In Japanese). [Online]. Available: http://codezine.jp/article/detail/6618

[4] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in Proc. 27th Annual Computer Security Applications Conference (ACSAC'11), pp.343-352, 2011.

[5] H. Kawabata, T. Isohara, K. Takemori, and A. Kubota, "Threat of script abuse android permissions and static analysis," IPSJ SIG Technical Report (In Japanese), vol.2011-CSEC-53, no.3, pp.1-6, 2011.

[6] (2009, Jan.) lexanderA - WebView examples. [Online]. Available: http://lexandera.com/category/webview_examples/

[7] (2011, Dec.) Dedexer user's manual. [Online]. Available: http://dedexer.sourceforge.net/

[8] (2013, Jun.) dex2jar - Tools to work with android .dex and java .class files. [Online]. Available: http://code.google.com/p/dex2jar/

[9] (2012, Oct.) JD-GUI - Java Decompiler. [Online]. Available: http://java.decompiler.free.fr/?q=jdgui

[10] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in javascript contexts," in Proc. 31st International Conference on Distributed Computing Systems (ICDCS'11), pp.720-729, 2011.

[11] L. Meyerovich, and B. Livshits, "CONSCRIPT: specifying and enforcing fine-grained security policies for javascript in the browser," in IEEE Symposium on Security and Privacy (SP'10), pp.481-496, 2010.

[12] D. Barrera, H. Kayacik, P. Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in Proc. 17th ACM Conference on Computer and Communications Security (CCS'10), pp.73-84, 2010.

[13] H. Hao, V. Singh, and W. Du, "On the effectiveness of api-level access control using bytecode rewriting in android," in Proc. 8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'13), pp.25-36, 2013.

[14] S. Sakamoto, K. Okuda, R. Nakatsuka, and T. Yamauchi, "DroidTrack: tracking information diffusion and preventing information leakage on android," in Proc. 7th FTRA International Conference on Multimedia and Ubiquitous Engineering, Lecture Notes in Electrical Engineering (LNEE), vol.240, pp.243-251, 2013.