Journal of High Speed Networks 22 (2016) 15–34
DOI 10.3233/JHS-160534
IOS Press

Detecting injected behaviors in HTML5-based Android applications

Jian Mao [a,*], Ruilong Wang [a], Yue Chen [a] and Yaoqi Jia [b]
[a] *School of Electronic and Information Engineering, BeiHang University, Beijing, China*
[b] *School of Computing, National University of Singapore, Singapore, Singapore*

Abstract. HTML5-based mobile applications (or apps) are built by using standard web technologies such as HTML5, JavaScript and CSS. Due to their cross-platform support, HTML5-based mobile apps are getting more and more popular. However, similar to traditional web apps, they are often vulnerable to script-injection attacks. It results in new threats to code integrity and data privacy. Compared to traditional web apps, HTML5-based mobile apps have more possible channels to inject code, e.g., contacts, SMS, files, NFC, and cameras. Even worse, the injected scripts may gain much more powerful privileges from the mobile apps than those in the traditional web apps.

In this paper, we propose an approach to detect injected behaviors in HTML5-based Android apps. Our approach monitors the execution of apps, and generates behavior state machines to describe the apps' runtime behaviors based on the execution contexts of apps. Once code injection happens, the injected behaviors will be detected based on deviation from the behavior state machine of the original app. We prototyped our approach and evaluated its effectiveness using existing code injection examples. The result demonstrates that the proposed method is effective in code injection detection for real-world HTML5-based Android apps.

Keywords: Android security, HTML5 apps, injected behaviors, detection

1. Introduction

HTML5-based mobile apps use HTML5 and CSS to describe the user interface, and use JavaScript to build the programming logic. Because the apps run in browser environments, e.g., WebView of Android [49] and UIWebView [47] of iOS, porting HTML5-based mobile apps from one platform to another becomes easy. As a result, HTML5-based apps are popular in cross-platform development. In order to provide system-level functionalities, e.g., accessing Camera and GPS, HTML5-based apps use middleware frameworks, such as PhoneGap [36], to access system resources. However, these web technologies brought new security challenges to mobile platforms. For example, the Cross-Site Scripting (XSS) attacks [52] is a typical channel to inject malicious code into web applications. In mobile devices, there are much more possible channels for such code-injection attacks, e.g., contacts, SMS, file systems, NFC, and cameras [6,12,29]. In addition, the injected code in Android shares the same system privileges with the victim app, resulting in more powerful attacks compared to that in web applications.

There are mainly two types of existing solutions to enhance the security of HTML5-based mobile apps. One type of solutions improves the permission mechanism of Android [24,30]. They try to limit privileges of untrusted code. However, in code-injection attacks, malicious code may be injected into trusted apps, where attackers' code is not constrained. The other type of solutions detects code-injection attacks by filtering code from data in the potential code-injection channels of the devices [28,29]. However, it does not provide comprehensive protection and only concentrates on the known injection channels. Along with the development of mobile devices, new channels may be exploited to inject malicious code.

We have the following observation about injected malicious code in HTML5-based apps: a single malicious behavior (such as accessing GPS location) caused by the injected code may also appear as normal functionality

[*]Corresponding author. E-mail: maojian@buaa.edu.cn.

in the vulnerable app. However, it usually happens in a different *program context*. Defining and identifying such contexts will be the key to detect injected malicious behaviors.

In this paper, we propose a new approach to detect malicious behaviors in HTML5-based Android apps. Our approach can accurately capture the contexts under which app behaviors take place. Based on apps' behaviors and their corresponding contexts, we build behavior state machines for HTML5-based Android apps. The behavior state machines can then be used to detect injected behaviors in these apps. We prototyped our solution in Android system, and evaluated its effectiveness with real-world HTML5-based Android apps.

Contributions. In summary, we make the following contributions:

- We propose an approach for detecting injected behaviors in HTML5-based Android apps based on new behavior state machines of apps.
- To build the behavior state machines for HTML5-based Android apps, we identify the relationship among the apps' behaviors and the program contexts where the behaviors take place during the execution of the apps.
- We prototype our approach, and evaluate our approach with real-world HTML5-based Android apps.

2. Background

2.1. WebView and PhoneGap

In Android, WebView uses the WebKit rendering engine to display web pages. It packages the web-browsing functionalities into a class. This class is the basis upon which apps can roll their own web browser or simply display some online contents within their activities [49].

Since WebView is designed to display web contents, which usually come from untrusted external sources, the Android Browser isolates it inside a sandbox. The sandbox prevents the JavaScript code in the WebView from accessing local system resources, such as contact lists, cameras, file systems, etc. To allow HTML5-based apps to have system-level accesses, WebView adds a bridge between the JavaScript code and the native Java code such as an API called "addJavascriptInterface()". In this way, it is possible for JavaScript code to invoke the outside native code, which is not restricted by WebView's sandbox and can access system resources.

Middleware frameworks (e.g., PhoneGap [36], RhoMobile [40], Appcelerator [5]) can provide this kind of bridges to web pages which run in WebView. In this paper, we focus on PhoneGap, which is the most popular one and supports most of the mobile platforms such as Android, iOS, Windows Phone.

The PhoneGap apps are hybrid. On one hand, the layout rendering of apps is done via web views like web apps; on the other hand, all code including HTML5, JavaScript and CSS, is packaged as apps for distribution and has access to native system resources [37].

PhoneGap can be extended with native plugins that allow developers to add functionalities for apps. These plugins can be called from JavaScript code, making direct communication between the native system and the HTML5-based apps. The overview of PhoneGap architecture is illustrated in Fig. 1.

By default, PhoneGap includes 16 basic plugins which allow apps to access to the device's Accelerometer, Camera, Compass, File System, etc. If they cannot meet the apps requirements, developers can either write their own plugins or use third-party plugins (so far, the total number is 1,014). The plugins provide a set of uniform JavaScript libraries that can be invoked by the apps JavaScript code. When an app needs to access system resources, it calls the APIs provided in the libraries, which will then invoke the Java code of PhoneGap, and eventually access the corresponding system resources.

So far, PhoneGap supports most of mobile operating systems, e.g., Apple iOS [27], BlackBerry [9], Google Android [4], LG webOS [50], Microsoft Windows Phone (7 and 8) [54], Nokia Symbian OS [43], Tizen (SDK 2.x) [44], Bada [7], Firefox OS [22] and Ubuntu Touch [46].

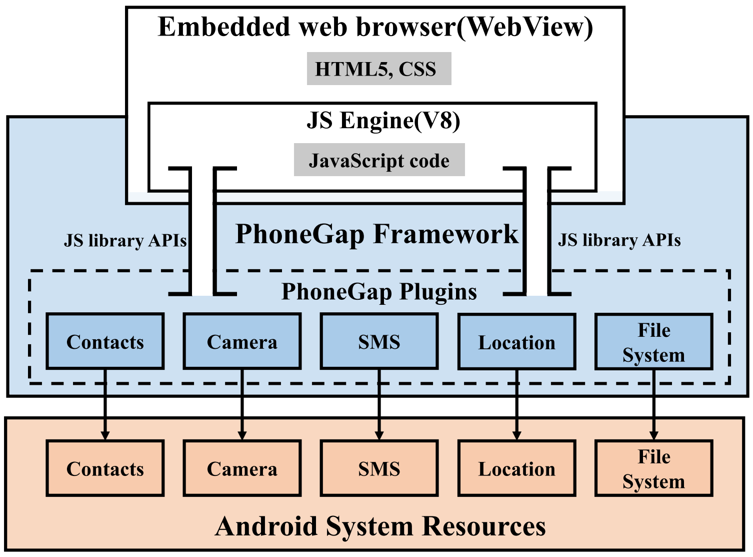


Fig. 1. The PhoneGap architecture on Android.

2.2. *Code injection attacks in web applications*

Web apps often make use of JavaScript code that is embedded into web pages to support dynamic client-side behaviors. This script code is executed in the users' web browsers. To protect users' system from the outside untrusted JavaScript code, access control such as the same-original policy (SOP) [53] is indispensable. The essential of SOP is that if contents from one site is granted permissions to access resources on the users' system, then any content from that site will share these permissions, while contents from other sites will have to be granted permissions separately.

Unfortunately, the XSS vulnerability enables attackers to bypass these security mechanisms to inject malicious code into web applications. Attackers can hide malicious code in the contents from the trusted site. When the contents containing the malicious code arrive at users' browsers, the injected code will get full access to all resources (e.g., authentication tokens and cookies) that belong to the trusted site. For example, attackers can hide the malicious code into an innocent-looking URL, clicking the link can cause the victim's browser to execute the injected script.

In recent years, XSS has surpassed buffer overflow [51] to become the most common publicly reported security vulnerability [48]. Some famous sites affected by XSS include Twitter, Facebook, MySpace, YouTube and Orkut [52].

2.3. *Code injection attacks in HTML5-based mobile apps*

With the middleware framework represented by PhoneGap, web apps can be easily ported to the mobile system, such as Android. Unfortunately, the code injection attack is also introduced into the mobile system. What is worse, mobile devices have more possible channels for attackers to inject code into the apps. Previous work [28,29] have already extensively discussed this serious problem.

Wi-Fi access point. Most of today's mobile devices can connect to Internet via Wi-Fi technology. Attackers can inject malicious JavaScript code into the field of a Wi-Fi access point's Service Set Identifier (SSID). If an HTML5-based mobile app scans the available Wi-Fi hotspots nearby and displays their SSIDs, the malicious code injected into the SSID will be compiled and executed in the victim's mobile device.

Barcode. Most mobile devices can scan barcode via cameras and certain apps. However, such functionality can be leveraged by attackers to inject code into mobile devices. Because the data inside the barcode are invisible to users, malicious JavaScript code can be embedded in the barcode data, which might be injected into apps due to the lack of sanitization checking.

SMS message. Attackers can also inject JavaScript code into the body of an SMS message. When this malicious SMS message is displayed in an HTML5-based app, the injected JavaScript code can be executed.

In this paper, we do not concentrate on the specific channel where the code been injected into the apps. We believe that new channels may emerge with the rapid development of mobile devices while the traditional methods cannot face that challenge. Instead, we focus on the specific effects caused by the injected code. App's behaviors and their program contexts will give us sufficient information to detect code injection attacks.

3. Design

3.1. Behavior state machines for HTML5-based Android apps

We propose a behavior state machine for HTML5-based Android apps. It includes two dimensions: *actions* and their *contexts*.

Actions. In our behavior state machine, actions denote behaviors of HTML5-based Android apps. We classify apps' actions into two categories: local actions and network actions.

- *Local Actions* are conducted by calling APIs provided by middleware frameworks, such as PhoneGap, to access local system resources (e.g., contact list, GPS coordinate, camera, file system, and so on).
- *Network Actions* are communications between the apps and remote servers. They are conducted as HTTP requests sent by the app.

Interface context. The interface contexts of an app may provide important logic clues to determine whether the app's behaviors are abnormal. Therefore, we use the apps' interfaces to represent the contexts of apps' behaviors. Due to their complexity, HTML5-based Android apps contain more types of interfaces than native Android apps. We classify the interfaces of HTML5-based Android apps into three categories, namely, *activity interface*, *HTML interface* and *jQuery interface*.

- *Activity interface*: For native Android apps, each interface is corresponding to an activity. If the app wants to present a new interface to users, it has to create a new activity. But in HTML5-based Android apps, which execute inside the built-in web browser, the graphic interface is developed using web technologies such as HTML5, CSS3, etc. So this kind of apps only needs one activity to create a WebView object, and uses different HTML files to present different graphic interfaces instead.
- *HTML interface*: As described above, HTML5-based Android apps use web programming languages to present user interfaces and achieve apps' functionalities. So different interfaces are represented by different HTML pages loaded in one WebView object.
- *jQuery interface*: jQuery Mobile is a HTML5-based user interface system designed to make responsive web sites and apps that are accessible on all smartphones, tablets and desktop devices [31]. Developers can create multiple pages as interfaces in a single HTML file using jQuery mobile technology. They can also allocate a unique ID for each page and use the "href" attribute of archor elements to switch interfaces.

3.2. Approach overview

The architecture overview of our system is illustrated in Fig. 2, there are four major components:

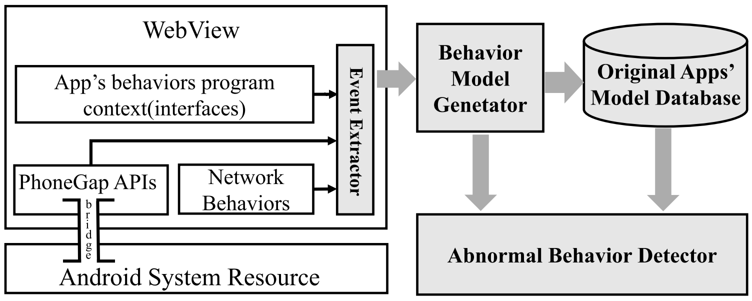


Fig. 2. Architecture overview.

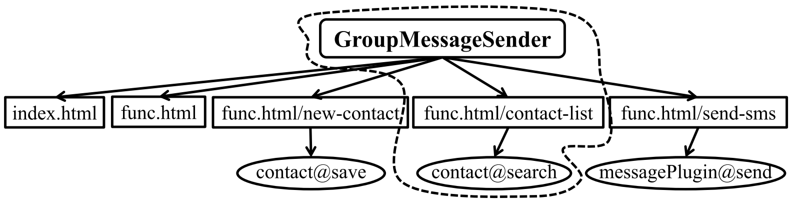


Fig. 3. GroupMessageSender's primitive behavior state machine.

Event extractor. "Event extractor" is a run-time module inside the Android system. By hooking into the WebKit component of the Android system, we extract necessary information about apps' actions and the program contexts where the actions take place. With our extractor, when the app makes a PhoneGap API call, sends an HTTP request, or creates a new interface, our system will record them.

Behavior state machine generator. According to the app's actions and the related interfaces extracted by the first module, "Behavior State Machine Generator" will automatically build the behavior state machine of the app. For every interface context, this module associates it with actions taking place in it.

Figure 3 presents a sneak preview of a app's behavior state machine. Our behavior state machine includes the app's actions (represented by ovals) and the pages where the actions take place (represented by rectangles). The pages represent the distinctive interface contexts of the actions. In the state machine, and the arrowed line from a page to an action illustrates the dependency between the context and the action. As shown in Fig. 3, there are four interface contexts and three actions in the app "GroupMessageSender". For every interface, if there is an arrowed line from it to an action, it means this interface is the distinctive context of the action. For example, in Fig. 3, page "index.html/scan" is the interface context of the action "exec@camera".

Original app's state machines database. Security analysts can run the app in a pure environment to obtain its original behavior state machine. After traversing the app and triggering all behaviors from each interface using a test suite, the output from the Behavior State Machine Generator will be treated as the app's original behavior state machine and stored in the behavior state machine database.

Abnormal behavior detector. In this module, we check the context of the app actions of the target app's behavior state machine against its original behavior state machine stored in the database. Specifically, given an action in an app's behavior state machine, we first identify its interface context. In the corresponding context, we check whether the action–context pair appears as a part in the original state machine to detect the abnormal behavior. If the pair of the action and its context is not in the original behavior state machine, it will be treated as an abnormal behavior. The result of detection not only reveals traces of the abnormal behaviors caused by the injected code, but also locates the specific page where the code injection attack happened.

4. Implementation

We implemented a proof-of-concept prototype of our solution in Android 4.1.2. Our system consist of 4 components: a run-time events extractor, a behavior state machine generator, a database for the original behavior state machines and an abnormal behavior detector.

The Event Extractor monitors the interfaces created by the app and the APIs called by the app (apps' actions and their corresponding APIs are illustrated in Table 1). In the WebKit part of Android source code, we set hook functions to get the apps' interfaces and actions information.

More specifically, for the app's *local actions* triggered by calling certain PhoneGap APIs, we set the hook function "WebCore_npObjectInvokeImpl" in the WebKit component to hook the APIs as well as the corresponding system resources that the app called. For the app's *network actions*, we set the hook function "XMLHttpRequest_createRequest" in the WebKit component where the XML HTTP requests are generated. For the app's *interfaces*, we set the hook function "WebFrame_documentLoaded" in the part of WebKit which creates the web frame according to the app's source code. We write all the hooked information into a log file and pass the file to the "Behavior State Machine Generator".

The Behavior State Machine Generator uses the log file generated by the Event Extractor to build the behavior state machines for the app. It allocates an array for every interface in the app, uses the interface's identifier as the first element of array, and adds the actions belong to the interface to the array as other elements. Then, it collects all interfaces' arrays in a set to form the final behavior state machine of the app.

Table 1

Apps' actions and the corresponding APIs

App's behaviors	The corresponding APIs
Access to the device's default camera application.	camera.getPicture, camera.cleanup
Access to the device contacts database.	contacts.create, contacts.find, contacts.remove
Read the device's information.	device.name, device.cordova, device.platform, device.uuid, device.version, device.model
Captures device motion in all directions.	accelerometer.getCurrentAcceleration, accelerometer.watchAcceleration, accelerometer.clearWatch
Access to the audio, image, and video capture capabilities of the device.	capture.captureAudio, capture.captureImage, capture.captureVideo, MediaFile.getFormatData
Send HTTP request.	GET@request's URL
Obtains the direction that the device is pointing.	compass.getCurrentHeading, compass.watchHeading, compass.clearWatch, compass.watchHeadingFilter, compass.clearWatchFilter
Access to the device's cellular and Wi-Fi connection information.	navigator.connection.type
Access to the device's GPS sensor.	geolocation.getCurrentPosition, geolocation.watchPosition, geolocation.clearWatch
Send SMS message.	MessagePlugin.send

5. Evaluation

In order to evaluate the effectiveness of our solution in detecting injected behaviors in HTML5-based Android apps, we deploy our solution to model the behaviors of real-world popular HTML5-based Android apps, and demonstrate its capabilities in detecting injected behaviors with simulated code injection attacks on those apps.

So far, we have collected 150 popular HTML5-based Android apps which download times are over 5,000 in GooglePlay. We also generated original behavior state machines for these apps. Among these 150 apps, we manually selected 4 vulnerable apps which have channels for attackers to inject code into them. We use these 4 apps as case studies to prove the effectiveness of our solution on injected behaviors detection. We simulated attackers' efforts to inject malicious JavaScript code into the apps. With the original behavior state machines we already built, we can successfully detect the injected behaviors in the apps.

There are various channels the mobile devices provided to attackers to inject malicious JavaScript code into the apps. In our case studies, the malicious JavaScript code is injected into a QR code and the name of a contact. When the app scans the QR code or reads the contact list, the injected code will be executed.

5.1. Case study: RewardingYourself

RewardingYourself [39], shown in Fig. 4, is an HTML5-based app that maintains the record of miles or points of the user's loyalty program. The app can take QR code as its input via a third-party barcode-scanner plugin. This additional input channel opens a door for code injection attack. Attackers can write QR code containing malicious JavaScript and when the app scans this QR code, the malicious code will be injected into the app.

This app is a case study in the paper of Wenliang Du et al. [29]. In their work, they can find the vulnerable channel (QR code) the app contains for the code injection attack. But, the result of the attack is beyond of their ability. Instead, our system can detect the injected behaviors brought by the malicious QR code. Furthermore, we can also know the detail of behaviors and their program contexts.

In this case study, we thoroughly test the app and explores its functionalities. Under the monitor of "Event Extractor" in our system, we get the record of the app's actions and the program contexts where the actions take place. "Behavior State Machine Generator" then use the record from "Event Extractor" to automatically build the original behavior state machine for RewardingYourself shown in Fig. 5. In Fig. 5, rectangles are interface contexts represented by different interfaces in the app, ovals denote app's actions, and the arrow from a context to an action denote the dependency between the app's action and it's interface context.

We use a malicious QR code, shown in Fig. 6, which embeds the malicious code shown in Fig. 7. The result of the attack is illustrated in Fig. 8.

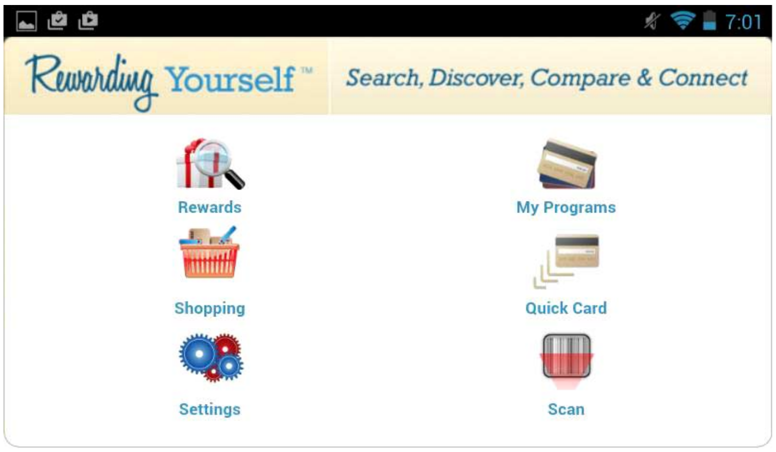


Fig. 4. The RewardingYourself application.

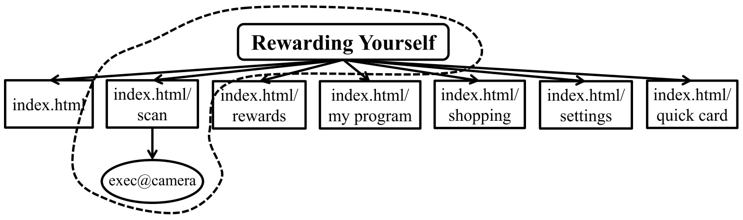


Fig. 5. Original behavior state machine of RewardingYourself.

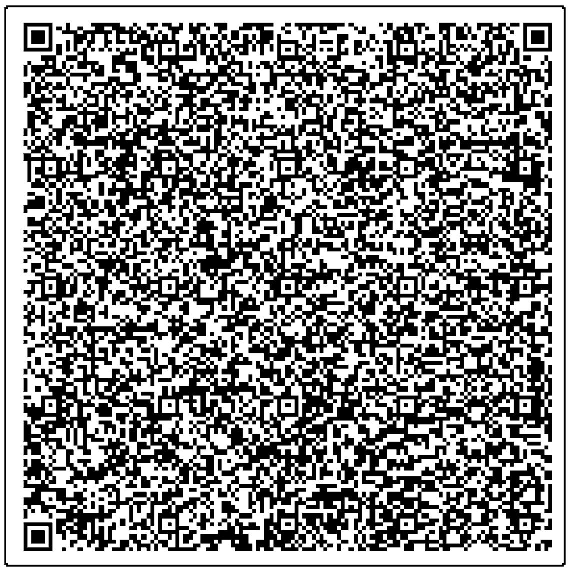


Fig. 6. QR code containing malicious JavaScript.

As it is illustrated in Fig. 9, because of the attack, new actions “position:1.295037, 103.773856” and `"Get@ http://172.26.191.206/info.php?msg=posit_ion@1.295037@103.773856@true"` appear under their interface context `"index.html/scan"`.

After checking the app’s original behavior state machine, we can detect the injected behaviors brought by the scanning of malicious QR code. The injected code will leak the current position of the user which is not merely a privacy leakage. Our approach can successfully detect the malicious behaviors brought by the injected code.

5.2. *Case study: TripCase*

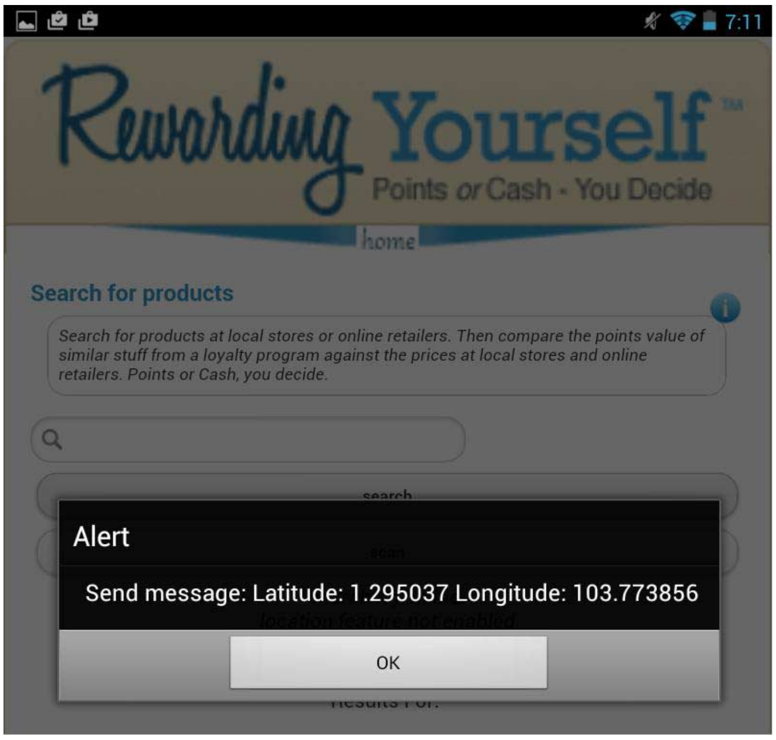
TripCase [45] is a travel management applications based on web technologies, as shown in Fig. 10. Users can organize their travel through it, for example set flight time alert, note the important places, organize the whole travel schedules, and so on. Because of the portability convenience the HTML5 technology provided, the trip information stored in TripCase is available almost all platforms such as phone, tablet, Android Wear and desktop website. TripCase is a popular travel assistant application, it is downloaded over 500,000 times in GooglePlay (the official app market in Android) and get recommendations from Forbes Travel Guide and The Washington Post.

Due to web technologies it use, TripCase share the vulnerabilities of HTML5-based Android apps. The various channels mobile devices provided make the app become a target of attackers to inject malicious code in it.

We deployed our solution to evaluate the effectiveness of injected behaviors detection with TripCase. Like RewardingYourself, we thoroughly test the app and explores its functionalities in our system. Because there is 11

```
<img src="x" onerror ='
var options = {maximumAge:30000, timeout:100000, enableHighAccuracy:false};
navigator.geolocation.watchPosition( //Get user’s current position
function onSuccess(position) {
var pos = "longitude:" + position.coords.longitude + "latitude:" +
          position.coords.latitude;
xmlhttp = new XMLHttpRequest(); //Send the position to remote server
xmlhttp.onreadystatechange = function() {if (xmlhttp.readyState==4 &&
xmlhttp.status==200) {alert('send message: ' + pos);}};
xmlhttp.open('GET', 'http://172.26.191.206/info.php?pos=' + pos, true);
xmlhttp.send();},function(){},options);'>
```

Fig. 7. The injected malicious code that attempts to read users current position and send it to the remote server.



Search for products at local stores or online retailers. Then compare the points value of similar stuff from a loyalty program against the prices at local stores and online retailers. Points or Cash, you decide.

Alert

Send message: Latitude: 1.295037 Longitude: 103.773856

OK

Fig. 8. The result of code injection attack to RewardingYourself.

interface contexts in the app, for demonstration purpose, we only illustrate a fragment of the compact behavior state machine in Fig. 11.

Up to now, the behavior state machine we built is under a trusted environment, we can save it as the primitive behavior state machine of TripCase in the database. Then, we simulated attackers’ efforts to inject malicious JavaScript code into the app. The malicious JavaScript code we injected is illustrated in Fig. 12.

We inject the attack code into a meeting’s address in TripCase, when the app read the malicious “address” we input, the injected code will be executed. The result of the execution will read the whole contact list of the victim user and send them to a specific remote server. The detail of the attack is illustrated in Fig. 13.

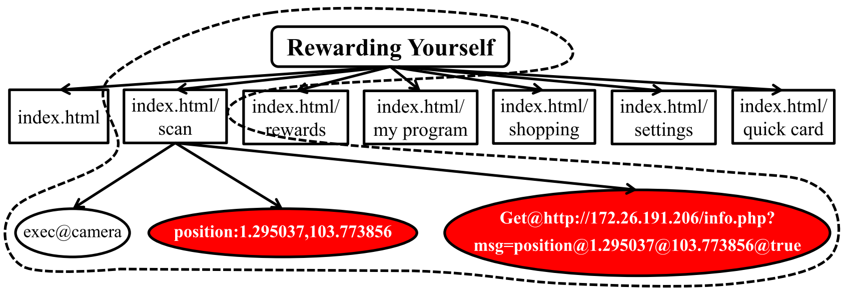
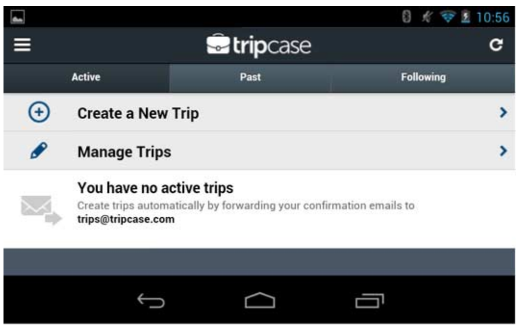


Fig. 9. Injected behaviors state machine in RewardingYourself.



You have no active trips

Create trips automatically by forwarding your confirmation emails to **trips@tripcase.com**

Fig. 10. The TripCase application.

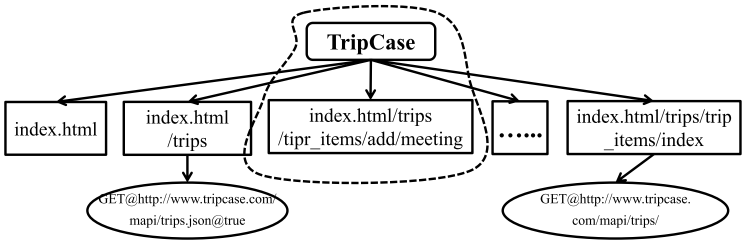

TripCase
index.html
index.html /trips
index.html/trips /tipr_items/add/meeting
......
index.html/trips/trip _items/index
GET@http://www.tripcase.com/ mapi/trips.json@true
GET@http://www.tripcase. com/mapi/trips/

Fig. 11. A fragment of original behavior state machine of TripCase.

Because the code injection happened, there are new actions (read contact list and send them to the server) under certain interface contexts. As it is illustrated in Fig. 14, under the same interface `"index.html/trips/trip_items/add/meeting"`, the compromised behavior state machine contains new actions "contact search" and `"Get@http://172.26.191.206/info.php?msg=Allen,+6591352163.Bob,+6592546547.Carl+6591877142.@true"`.

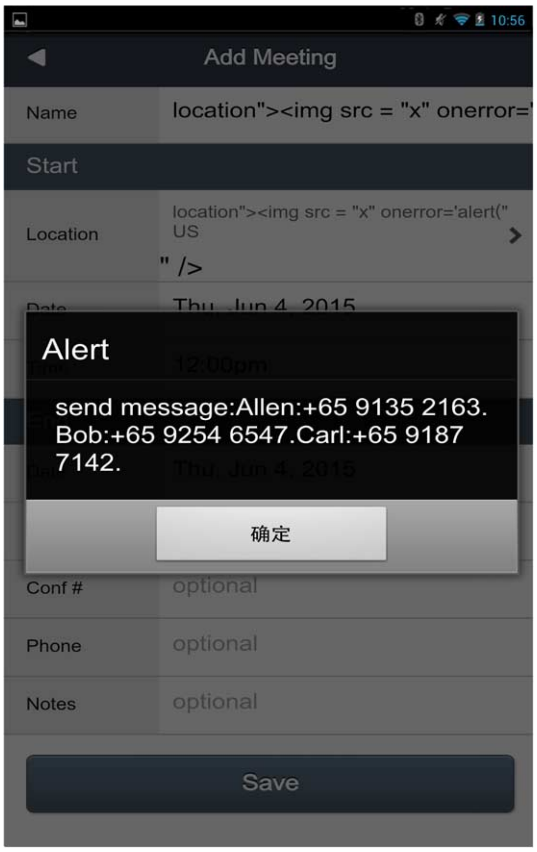
After checking the app's original behavior state machine we can successfully detect the injected behaviors in the compromised app. Furthermore, we can also know the specific page where the code injection attack happened, which gives security analysts a clue to find vulnerabilities of the app.

5.3. *Case study: PRS Barcode Scanner*

PRS Barcode Scanner [38] shown in Fig. 15 is a quite popular barcode scanner app based on the HTML5 technology. It is downloaded almost 5,000 times in GooglePlay.

```
<img src=“x” onerror=“var fields = [‘*’];
var options = new ContactFindOptions();   //Read user’s contact list
options.multiple = true;
navigator.contacts.find(fields,
function(contacts) {
var msg = '';
for (var i = 0; i <= contacts.length - 1; i ++) {
msg += contacts[i].displayName + ', ' + contacts[i].phoneNumbers[0].value +
       ', ' + contacts[i].emails[0].value + '.';}
xmlhttp = new XMLHttpRequest();   //Send the contact list to remote server
xmlhttp.onreadystatechange = function() {
if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
alert(xmlhttp.responseText);}}
xmlhttp.open('GET', 'http://172.26.191.206/info.php?msg=' + msg, true);
xmlhttp.send();},
function() {},options);}"/>
```

Fig. 12. The injected malicious code that attempts to read user’s contact list and send it to remote server.



location"><img src = "x" onerror='

location"><img src = "x" onerror='alert("
US
" />

Thu, Jun 4, 2015

Alert

send message:Allen:+65 9135 2163. Bob:+65 9254 6547.Carl:+65 9187 7142.

Fig. 13. The result of the attack to TripCase.

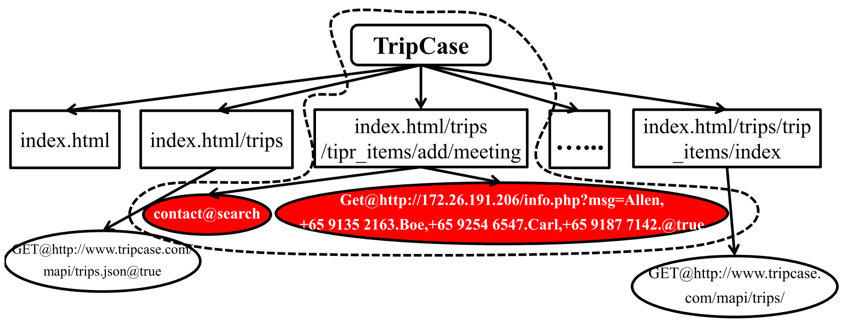

TripCase
index.html
index.html/trips
index.html/trips /tipr_items/add/meeting
……
index.html/trips/trip _items/index
contact@search
Get@http://172.26.191.206/info.php?msg=Allen, +65 9135 2163.Boe,+65 9254 6547.Carl,+65 9187 7142.@true
GET@http://www.tripcase.com/ mapi/trips.json@true
GET@http://www.tripcase. com/mapi/trips/

Fig. 14. Injected behaviors state machine in TripCase.

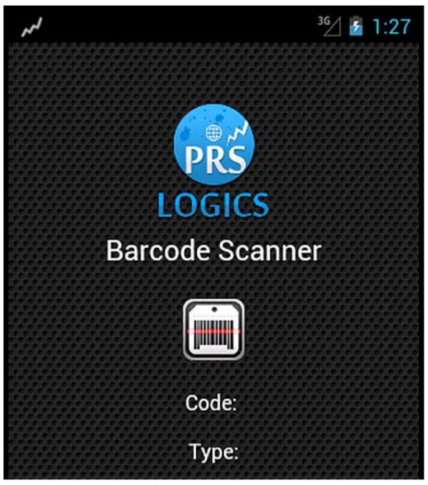


Fig. 15. The PRS Barcode Scanner application.

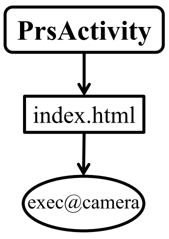
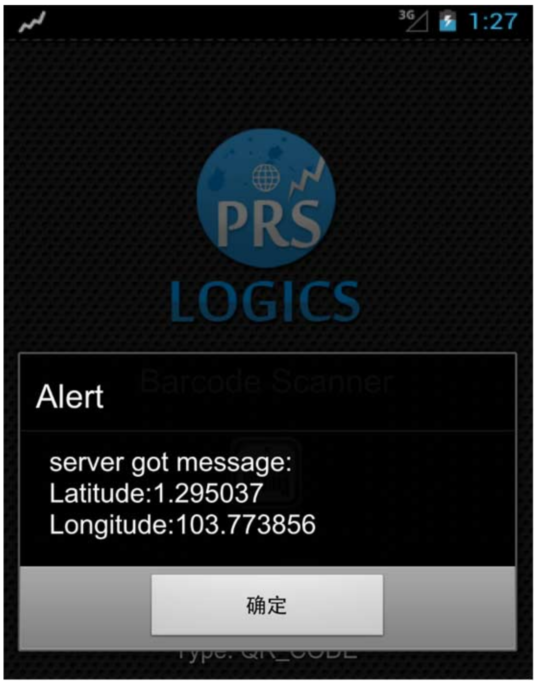


Fig. 16. Original behavior state machine of PRS Barcode Scanner.

It share the same vulnerabilities of HTML5-base Android apps. We also use the malicious QR code (as it is illustrated in Fig. 6) to let the app scan. Under our system, we can successfully detect the injected behaviors brought by the scanning of malicious QR code.

We also traverse the app in our system, and get the original behavior state machine of it, as it is illustrated in Fig. 16. Then, we simulated attackers' efforts to inject malicious JavaScript code into the app. The result of the attack is illustrated in Fig. 17.

Because of the attack, there are new actions under certain interface contexts. As it is illustrated in Fig. 18, under the same interface context "`index.html/scan`", the compromised behavior state machine contain new actions



server got message:
Latitude:1.295037
Longitude:103.773856

Fig. 17. The result of code injection attack to PRS Barcode Scanner.

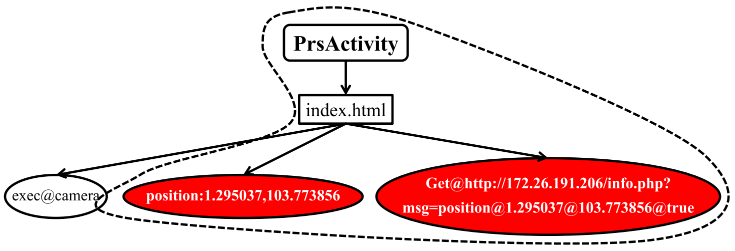

PrsActivity
index.html
exec@camera
position:1.295037,103.773856
Get@http://172.26.191.206/info.php?
msg=position@1.295037@103.773856@true

Fig. 18. Injected behaviors state machine in PRS Barcode Scanner.

`"position:1.295037,103.773856"` and `"Get@http://172.26.191.206/info.php?msg=position@1.295037@103.773856@true"`. After checking the app's original behavior state machine we can successfully detect the injected behaviors in the compromised app.

5.4. *Case study: PhoneGapMega*

PhoneGapMega [35] is a demonstration app for PhoneGap APIs shown in Fig. 19. It contains examples of almost all of the PhoneGap APIs (16 kinds of them) which the framework provide to developers by default. jQuery Mobile is used as the display framework in the app.

PhoneGapMega is a popular PhoneGap example app in GooglePay which has been downloaded almost 10,000 times. It is just a representative of that kind of PhoneGap APIs example apps, there are other apps which serve as the same purpose with PhoneGapMega such as cordova-mega-demo [15], GWT Mobile PhoneGap Showcase [25], PhoneGap Demo [34], and so on.

PhoneGapMega share the same vulnerabilities of HTML5-base Android apps. Malicious JavaScript (shown in Fig. 12) injected into a contact's name can be executed in the app.

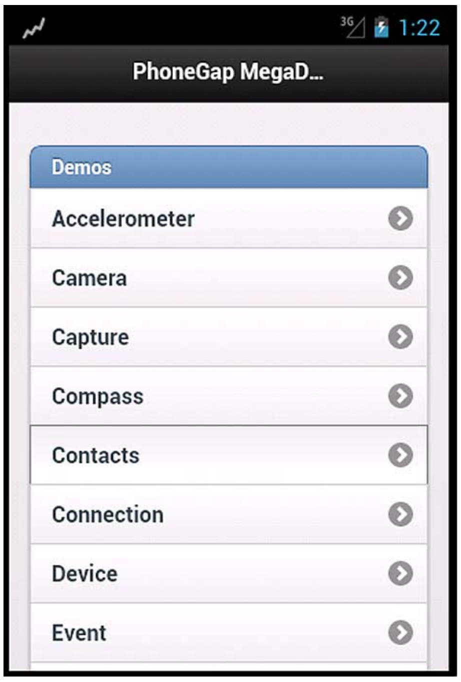


Fig. 19. The PhoneGapMega application.

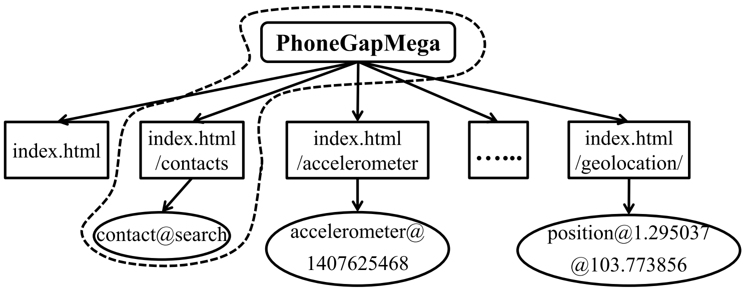


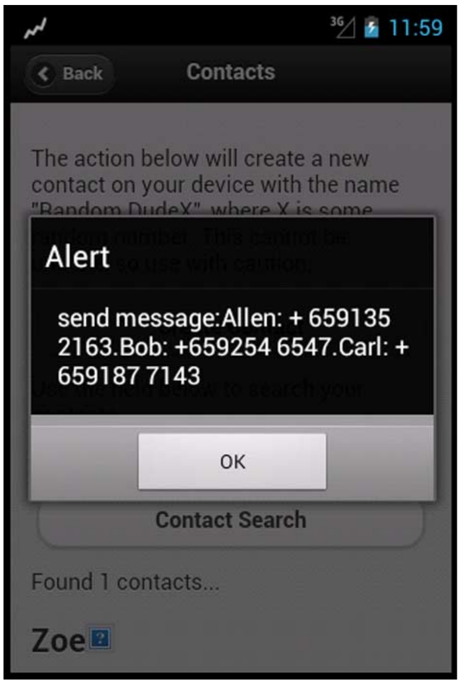
Fig. 20. A fragment of the original behavior state machine of PhoneGapMega.

We also traverse all interfaces in the app and explore every action belong to each interface. Because PhoneGapMega contains almost all of 16 default PhoneGap APIs and allocate one page for each API, for demonstration purpose, we only illustrate a fragment of the compact behavior state machine in Fig. 20.

When the app read the malicious contact item, the injected code will be executed. The result of the execution will read the whole contact list of the victim user and send them to a specific remote server, as it is illustrated in Fig. 21.

Because of the attack, there are new actions under certain interface contexts. As it is illustrated in Fig. 22, under the same interface context `"index.html/contacts/index.html"`, the compromised behavior state machine contains new actions "contact search" and `"Get@http://172.26.191.206/info.php?msg=Allen,+6591352163.Bob,+6592546547.Carl+6591877142.@true"`. Finally, we successfully detect the injected behaviors brought by the injected code.

Summary. With the above case studies, we demonstrate the importance of including app's actions and program contexts where actions take place in behavior state machine generation. It confirms that our approach is effective in detection of injected behaviors in HTML5-based Android apps.



11:59

The action below will create a new contact on your device with the name "Random DudeX" where X is some

Alert

send message:Allen: + 659135 2163.Bob: +659254 6547.Carl: + 659187 7143

Found 1 contacts...

Zoe

Fig. 21. The result of code injection attack to PhoneGapMega.

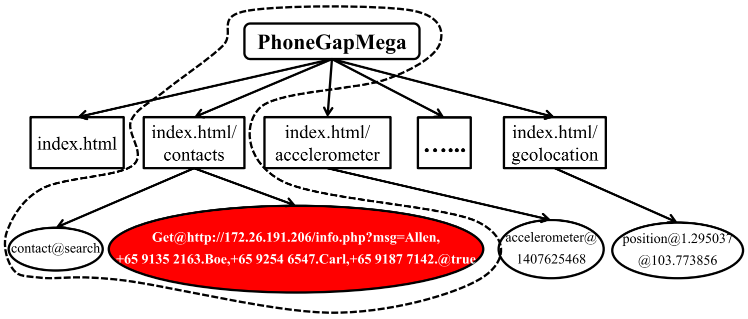

PhoneGapMega
index.html
index.html/ contacts
index.html/ accelerometer
……
index.html/ geolocation
contact@search
Get@http://172.26.191.206/info.php?msg=Allen, +65 9135 2163.Boe,+65 9254 6547.Carl,+65 9187 7142.@true
accelerometer@ 1407625468
position@1.295037 @103.773856

Fig. 22. Injected behaviors state machine in PhoneGapMega.

5.5. *Performance evaluation*

To measure the performance overhead of our system, we benchmarked it against original Android system. Table 2 lists the average page loading time for above case studies apps, measured on a Linux desktop with Intel Core i3-2130 CPU with 4 GB of RAM.

As it is shown in Table 2, our system introduces approximately 0.108 s (6.75%) additional time cost in page loading compared to original Android system, which is hard for users to perceive it.

So far, we have generated original behavior state machines for 150 popular HTML5-based Android apps,[1] whose downloads are over 5,000 in GooglePlay (selected apps are listed in Table 3).

[1] All of these apps are downloaded from GooglePlay.

Table 2
Performance overhead of our system

Experiment environment	Linux desktop Intel Core i3-2130 CPU 4 GB of RAM	
Platforms	Original Android	Our system
Page loading time	1.599 s	1.707 s

Table 3
Apps' information in the original behavior state machine database

App name	Developer	Download times
TripCaser	Sabre Traveler Solutions	500k–1,000k
28th SEA Games TV	SINGSOC	100k–500k
Beauté-test	Beauté-test.com	50k–100k
Camera photo effects	FOTOEFECTOS.COM	50k–100k
NewsBook	Aming	100k–500k
Black Dating For Free	E Dating For Free, Inc.	100k–500k
Bhinneka.Com	Bhinneka Mentari Dimensi	100k–500k
AccesVPN	VPN Corporation	100k–500k
Discover Beer	Untappd	1,000k–5,000k
PBR Now	Professional Bull Riders, Inc.	50k–100k
News Reader	Netease	500k–1,000k
On The Snow Ski	SkiReport.com	1,000k–5,000k
Vehicle Sounds Lite	App Dev Team	50k–100k
LibraryAnywhere	LibraryThing.com	50k–100k
Connect the Dots	IPV Solutions	100k–500k
Inganna la mente	Paolo Tuttoilmondo	500k–1,000k
Music Tutor	JSplash Mobile	100k–500k
BloodPressureDB	Horst Klier	100k–500k
Farmacia di Turno	Visiant Contact	500k–1,000k
Gallery Lock	Sweet Sugar	100k–500k
MathOpen	MathOpen	500k–1,000k
Emlak Jet	emlakjet.com	100k–500k
WordArt	FlipFlox	100k–500k
Freecycle	trashnothing.com	50k–100k
Xaveco	dutonS.com	500k–1,000k
DomoSimu	DomofinanceEDF	5k–10k
Breathefree App	Cipla Digital	5k–10k
Baby Animal Puzzle Lite	App Dev Team	5k–10k
Hosting	Internet Tutorial & ARYACO	5k–10k
AteneoWeb Mobile	AICON	5k–10k

The original app's behavior state machines is generated by traversing the whole app and triggering all behaviors of the app. If there is a code injection attack in these apps, the malicious code must contain some "new" behaviors compared to the original apps.

As it is shown in above case studies, with the compact original behavior state machines, our system can successfully detect the injected behaviors. Typically, there is no false positive in our system consider the completeness of the original app's behavior state machines.

As far as we know, there is a limitation in our approach. If the malicious code conducts the same behaviors under the same contexts with the original app, we can not detect them. In our future work, we will try to find a more fine-grained definition of *program context* to solve this problem.

6. Related work

6.1. Existing researches on native Android apps

Control flow-based detection. The basic idea of control flow-based detection is to monitor and analyze the control flow of the Android apps [14,41,42], which includes system calls and their corresponding trigger events [3].

CrowDroid [11] is a framework to analyze smartphone apps' actions. CrowDroid monitors every system call of the apps and counts the number of each call the app has sent. The remote server will create a feature vector for the app based on crowdsourcing. If there is a malicious app that shares the name with a benign one, the system calls of the malicious app must be different from the benign one, as a result, the vector of the malicious app will be different. Base on the difference between the vectors, CrowDroid can detect anomalous behaviors of the malicious app counterfeiting the name and version of a benign one.

DroidSIFT [55] classifies Android malware base on weighted dependency graphs of apps' contextual APIs. According to the graph-similarity metrics, DroidSIFT may discover the homogeneous app behaviors and identify the transformation attacks and malicious variants.

Data flow-based detection. Data flow-based solutions are focused on users' privacy leakage detection and prevention. Jung et al. [32] propose a mechanism to find the information leakage with the black box differential testing. Felt and Evans [19] suggest returning fake data when apps require the user's privacy information to maintain normal operations and prevent the privacy leakage.

TaintDroid [17] classifies the privacy data in the Android system and assigns an unique *taint tag* to each data type. It tracks privacy data processing dynamically. When the tainted data is leaving the device, TaintDroid will record the Taint tag and the next-hop of the data.

AppFence [26] offers two different approaches to protect sensitive data in Android apps: shadowing sensitive data and blocking sensitive data from being exfiltrated off the device. It substitutes shadow data for data that the user wants to hide, and blocks network transmissions that contain sensitive data only for apps' on-device usage.

Permission-based detection. Permission-based detections aim at optimizing the Android system's permission management by setting granular permission rules or detecting the dangerous permissions set in apps [8,20,23,33].

XManDroid [10] extends the monitor mechanism to detect and prevent privilege escalation attacks on Android application level [16]. It sets a series of secure policies about Inter-component communication (ICC) [13] between two apps. It dynamically analyzes apps' transitive permission used by ICC call, and examines these two apps' permissions against the secure policy. If there is dangerous combination of permissions from these two apps, XManDroid will prohibit the ICC call between them.

TISSA [57] provides a fine-grained access control mechanism for users to grant the apps' accesses to different types of private information. The granted access may be changed on-demand according to the privacy requirements in specific scenarios. Kirin [18] is a security checker to identify the dangerous permissions by analyzing the apps' manifest files. It identifies permissions in the app's manifest file during the installation and checks whether they conform with security policies.

Beside above three kinds of detections, there are some works on detecting anti-forensics activities in Android [1,2]. Pietro Albano et al. prove that it could be possible to artificially create a false digital alibi, plausible in the context of a legal proceeding, by exploiting some features of an Android device. The proposed techniques make use of a software automation which is able to fully simulate a real series of human activities.

However, there are attacks that do not need special permissions. In [21], authors carried out energy-based attacks through the Android browser without installing App to users' Android system. Zhang et al. [56] proposed a new approach called App Guardian, which changes neither the operating system nor the target apps, and provides immediate protection as soon as an ordinary app is installed. App Guardian thwarts a malicious app's runtime monitoring attempt by pausing all suspicious background processes when the target app is running in the foreground, and resuming them after the app stops and its runtime environment is cleaned up.

6.2. *Existing research on HTML5-based Android apps*

Georgiev et al. [24] and Jin et al. [28] propose solutions to prevent untrusted foreign-origin web code from accessing local system resources. They modified the PhoneGap framework and extend an whitelist-based permission system. They limit the privilege of untrusted code and forbid them to access sensitive local resources.

Du et al. [30] propose a fine-grained access control mechanism to refine the privilege authorization policies for HTML5-based apps in Android system. They define two types of policies, frame-based and origin-based, to separate subjects of the same app. Developers can specify frame permissions using the permissions attribute for frames in HTML code, or origin permissions in the manifest file.

Besides the permission based protection, Du et al. [28,29] make a systematical study on the code injection attacks to the HTML5-based mobile apps and test different possible injection channels provided by the mobile devices. They also develop an automatic tool to detect code injection vulnerabilities in HTML5-based Android apps and provide a prototype called NoInjection as a patch to PhoneGap in Android. The core idea of NoInjection is to filter out the malicious code hidden in the data provided by the attack channels. But it cannot handle the situation that the code is dynamically loaded from remote servers.

7. Conclusion

Using web technologies (e.g., HTML5, JavaScript, CSS) to build HTML5-based mobile apps introduces new security challenges in mobile platforms. In this paper, we propose a new behavior state machine for HTML5-based Android apps, as well as an abnormal behavior detection system. The behavior state machine captures an app's actions as well as the contexts where actions take place, providing sufficient information about the program context of the app's behaviors. Our prototype detection system can automatically build the behavior state machines for HTML5-based Android apps, which will be compared with original one to detect abnormal behaviors. We demonstrate its effectiveness of abnormal behaviors detection with case studies on real-world HTML5-based Android apps.

Acknowledgements

The authors thank Zhenkai Liang for suggestions on various aspects of this paper. The authors also thank anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China (No. 61402029), the Beijing Natural Science Foundation (No. 4132056), the National 973 Project of China (No. 2012CB315905), the National Natural Science Foundation of China (Nos. 61170246, 376349), and the Beijing Natural Science Foundation (No. 4122024).

References

[1] P. Albano, A. Castiglione, G. Cattaneo, G. De Maio and A. De Santis, On the construction of a false digital alibi on the Android OS, in: *2011 Third International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, IEEE, 2011, pp. 685–690.

[2] P. Albano, A. Castiglione, G. Cattaneo and A. De Santis, A novel anti-forensics technique for the Android OS, in: *2011 International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, IEEE, 2011, pp. 380–385.

[3] S. Anand, M. Naik, M.J. Harrold and H. Yang, Automated concolic testing of smartphone apps, in: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM, 2012, Article No. 59.

[4] Android official site, available at: https://www.android.com/, January 2015.

[5] Appcelerator official site, available at: http://www.appcelerator.com/, February 2015.

[6] A. Armando, A. Merlo and L. Verderame, An empirical evaluation of the Android security framework, in: *Security and Privacy Protection in Information Processing Systems*, Springer, 2013, pp. 176–189.

[7] Bada official site, available at: www.bada.com, September 2014.

[8] D. Barrera, H.G. Kayacik, P.C. van Oorschot and A. Somayaji, A methodology for empirical analysis of permission-based security models and its application to Android, in: *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM, 2010, pp. 73–84.

[9] Blackberry OS official site, available at: http://www.blackberry.com/, November 2014.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer and A.-R. Sadeghi, XManDroid: A new Android evolution to mitigate privilege escalation attacks, Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[11] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani, CrowDroid: Behavior-based malware detection system for Android, in: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2011, pp. 15–26.

[12] A. Castiglione, R. De Prisco and A. De Santis, Do you trust your phone? in: *10th International Conference, EC-Web 2009*, September 2009, Springer.

[13] E. Chin, A.P. Felt, K. Greenwood and D. Wagner, Analyzing inter-application communication in Android, in: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ACM, 2011, pp. 239–252.

[14] M. Christodorescu, S. Jha and C. Kruegel, Mining specifications of malicious behavior, in: *Proceedings of the 1st India Software Engineering Conference*, ACM, 2008, pp. 5–14.

[15] cordova-mega-demo, available at: https://play.google.com/store/apps/details?id=au.com.redata.android.demo, April 2015.

[16] L. Davi, A. Dmitrienko, A.-R. Sadeghi and M. Winandy, Privilege escalation attacks on Android, in: *Information Security*, Springer, 2011, pp. 346–360.

[17] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L.P. Cox, J. Jung, P. McDaniel and A.N. Sheth, TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones, *ACM Transactions on Computer Systems (TOCS)* **32**(2) (2014), Article No. 5.

[18] W. Enck, M. Ongtang and P. McDaniel, On lightweight mobile phone application certification, in: *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ACM, 2009, pp. 235–245.

[19] A. Felt and D. Evans, Privacy protection for social networking APIs, in: *2008 Web 2.0 Security and Privacy (W2SP'08)*, 2008.

[20] A.P. Felt, E. Chin, S. Hanna, D. Song and D. Wagner, Android permissions demystified, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 2011, pp. 627–638.

[21] U. Fiore, F. Palmieri, A. Castiglione, V. Loia and A. De Santis, Multimedia-based battery drain attacks for Android devices, in: *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, IEEE, 2014, pp. 145–150.

[22] Firefox OS official site, available at: mozilla.org/firefox/os, March 2014.

[23] D. Gallingani, R. Gjomemo, V.N. Venkatakrishnan and S. Zanero, Static detection and automatic exploitation of intent message vulnerabilities in Android applications.

[24] M. Georgiev, S. Jana and V. Shmatikov, Breaking and fixing origin-based access control in hybrid web/mobile application frameworks, in: *NDSS Symposium*, Vol. 2014, NIH Public Access, 2014, p. 1.

[25] GWT mobile PhoneGap showcase, available at: https://play.google.com/store/apps/details?id=com.gwtmobile.phonegap, August 2014.

[26] P. Hornyack, S. Han, J. Jung, S. Schechter and D. Wetherall, These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications, in: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 2011, pp. 639–652.

[27] iOS official site, available at: https://www.apple.com/ios/what-is/, November 2014.

[28] 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*, ACM, 2014, pp. 66–77.

[29] X. Jin, T. Luo, D.G. Tsui and W. Du, Code injection attacks on HTML5-based mobile apps, arXiv preprint, 2014, available at: arXiv:1410.7756.

[30] 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.

[31] jQuery mobile official site, available at: http://jquerymobile.com/, March 2014.

[32] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis and T. Kohno, Privacy oracle: A system for finding application leaks with black box differential testing, in: *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ACM, 2008, pp. 279–288.

[33] 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*, ACM, 2010, pp. 328–332.

[34] PhoneGap demo, available at: https://play.google.com/store/apps/details?id=com.dreamappworld.android.phonegapdemo.free, September 2014.
[35] PhoneGapMega, available at: https://play.google.com/store/apps/details?id=com.camden.phonegapmega, June 2014.
[36] PhoneGap official site, available at: http://phonegap.com/, January 2015.
[37] PhoneGap on Wikipedia, available at: https://en.wikipedia.org/wiki/PhoneGap, June 2014.
[38] PRS Barcode Scanner, available at: https://play.google.com/store/apps/details?id=com.prs.barcodescanner, March 2015.
[39] RewardingYourself, available at: https://play.google.com/store/apps/details?id=com.loyaltymatch.rewardingyourself, May 2014.
[40] RhoMobile official site, available at: http://rhomobile.com/, April 2015.
[41] K. Rieck, T. Holz, C. Willems, P. Düssel and P. Laskov, Learning and classification of malware behavior, in: *Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2008, pp. 108–125.
[42] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer and Y. Weiss, "Andromaly": A behavioral malware detection framework for Android devices, *Journal of Intelligent Information Systems* **38**(1) (2012), 161–190.
[43] Symbian OS official site, available at: http://www.symbian.net/, January 2014.
[44] Tizen official site, available at: https://www.tizen.org/, March 2014.
[45] TripCase, available at: https://play.google.com/store/apps/details?id=com.sabre.tripcase.android, February 2014.
[46] Ubuntu Touch official site, available at: http://www.ubuntu.com/phone, December 2014.
[47] UIWebView in iOS, available at: http://developer.android.com/reference/android/webkit/WebView.html, July 2014.
[48] Vulnerability types distributions in CVE, available at: https://cwe.mitre.org/documents/vuln-trends/index.html, March 2015.
[49] WebView in Android, available at: https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/, October 2014.
[50] Wiki on LG webOS, available at: https://en.wikipedia.org/wiki/WebOS, January 2015.
[51] Wiki on the buffer overflow, available at: https://en.wikipedia.org/wiki/Buffer_overflow, October 2014.
[52] Wiki on the cross site scripting, available at: https://en.wikipedia.org/wiki/Cross-site_scripting#cite_note-9, February 2015, April 2015.
[53] Wiki on the same-origin policy, available at: https://en.wikipedia.org/wiki/Same-origin_policy, January 2015.
[54] Windows phone official site, available at: www.microsoft.com/windowsphone/, February 2014.
[55] M. Zhang, Y. Duan, H. Yin and Z. Zhao, Semantics-aware Android malware classification using weighted contextual API dependency graphs, in: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2014, pp. 1105–1116.
[56] N. Zhang, K. Yuan, M. Naveed, X. Zhou and X. Wang, Leave me alone: App-level protection against runtime information gathering on Android, in: *2015 IEEE Symposium on Security and Privacy (SP)*, 2015, pp. 915–930.
[57] Y. Zhou, X. Zhang, X. Jiang and V.W. Freeh, Taming information-stealing smartphone applications (on Android), in: *Trust and Trustworthy Computing*, Springer, 2011, pp. 93–107.