Fine-Grained Access Control for HTML5-Based Mobile Applications in Android

Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du*
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

ABSTRACT

HTML5-based mobile applications are becoming more and more popular because they can run on different platforms. Several newly introduced mobile OS natively support HTML5-based applications. For those that do not provide native support, such as Android, iOS, and Windows Phone, developers can develop HTML5-based applications using middlewares, such as PhoneGap [17]. In these platforms, programs are loaded into a web component, called WebView, which can render HTML5 pages and execute JavaScript code. In order for the program to access the system resources, which are isolated from the content inside WebView due to its sandbox, bridges need to be built between JavaScript and the native code (e.g. Java code in Android). Unfortunately, such bridges break the existing protection that was originally built into WebView.

In this paper, we study the potential risks of HTML5-based applications, and investigate how the existing mobile systems' access control supports these applications. We focus on Android and the PhoneGap middleware. However, our ideas can be applied to other platforms. Our studies indicate that Android does not provide an adequate access control for this kind of applications. We propose a fine-grained access control mechanism for the bridge in Android system. We have implemented our scheme in Android and have evaluated its effectiveness and performance.

1. INTRODUCTION

Mobile application (or mobile app) [13] is software application that runs on mobile devices, such as smartphones and tablets. In most mobile operating systems, mobile apps are written using a language chosen by the OS. For example, Android chooses Java, iOS chooses Objective C, and Windows Phone chooses C#. Applications written using the platform-selected language are called native mobile apps, because they are natively supported by the OS. Native mobile apps have advantages: they are more effective at integrating the unique features of the mobile device into apps, such as the telephone, voice recorder and camera. They also offer better user experience and performance. Unfortunately, the development of native mobile application is seen as expensive and laborious, because developers often need to support multiple platforms, and porting the code from one platform to another is not an easy task [11, 30].

HTML5-Based Mobile Applications. With the increasing support for HTML5, HTML5-based mobile applications are becoming more and more popular [7, 10, 19]. These applications are built on standard technologies such as HTML5, CSS and JavaScript, with HTML5 and CSS being used for building the graphical user interface, and JavaScript being used for the programming logic. Because these technologies are the basis of the Web, they are universally supported by all mainstream mobile systems. Porting such apps from one platform to another is much more simplified; of course, if they use platform-specific APIs, they need to be modified to run on a different platform, but this job is much easier than porting the native mobile applications [11, 35].

Due to HTML5-based mobile applications' advantages, most new mobile systems choose to support them as their native applications, instead of choosing their own platform-specific languages, like what Android, iOS, and Windows Phone did. This way, it will be much easier for developers to develop applications for them. Mozilla recently released its "made of the web" Firefox OS [6] in late February 2013, and Samsung released Tizen [20] operating system in 2012. They are designed to execute HTML5 applications directly in the system and allow them to communicate directly with the device's hardware using JavaScript.

The advantages of HTML5-based applications have also attracted people to use the same technology to develop applications for existing popular mobile platforms such as Android and iOS, so developers only need to develop one version of applications that can run on multiple platforms. Because the OS cannot natively support HTML5-based applications, middleware is needed for such applications to run on these platforms. Several such middlewares have been developed, including PhoneGap [17], RhoMobile [18], Appcelerator [2], WidgetPad [24], MoSync [14], etc. Because PhoneGap is the most popular one [16], in the rest of this paper, we use PhoneGap to represent this entire class of middlewares.

An essential technology needed for HTML5-based applications is web container, which renders HTML files and executes JavaScript program. Because all mobile devices need to access the Web, such a web container is built into almost all mobile operating systems. For Firefox OS and Tizen, web container is an integral part of the operating system. For Android, iOS, and Windows Phone, web container is a component that can be embedded into any application. In Android, the name of the component is `WebView`; in iOS, it is called `UIWebView`; in Windows Phone, it is called `WindowsBrowser`. Since we mainly focus on Android, we use WebView throughout this paper.

*This work is supported in part by NSF grants No. 1017771, No. 1318814, and by a Google research award.

Web container is designed to host web contents, but it is not sufficient to support HTML5-based mobile applications. Because of its purpose, web container allows its inside contents to only access the resources related to the Web (e.g. cookies, HTML5 local storage, etc.); many of the device resources are beyond the reach of the content inside web container. This is achieved by the sandbox built into all web containers; without it, contents from malicious web sites can pose great threats to the system. Unfortunately, this design makes it impossible to use the web container to host mobile applications, because these applications need to access device resources, such as camera, bluetooth, contact list, SMS, phone functions, etc. To solve this problem, a bridge has to be added to web container, allowing JavaScript code inside to access the native system resources.

The Emerging Threats. The bridges essentially create holes on the sandbox of web container. If these holes do not come with an adequate access control, they will soon become an area of active attacks. If we only use web container to hold trusted contents (such as HTML page from a local file), then these holes do not pose a problem, because on most platforms, there is another layer of sandbox, the application sandbox, which restricts the privileges of the application. Unfortunately, web containers are designed to host not only local HTML files, but also HTML files from remote web servers. Therefore, using web containers to host mobile applications adds a great risk that does not exist for native applications. We describe some potential attacks on HTML5-based mobile applications in the following.

HTML5-based mobile applications can easily include third-party components in themselves, and the code for these components may not be known during the installation, like advertisements, Facebook and Twitter social plugins [5,21]. In Android's and iOS's native application scenario, the source of the app and its corresponding dynamic code is known and trusted by the developers. Besides that, digital signature on the code adds a level of trustworthiness to the code. For HTML5-based mobile applications, programs are written in JavaScript. Due to the nature of the Web, JavaScript code can be loaded into applications dynamically. For example, an HTML5-based application may use an `iframe` to load advertisement, which is fetched from a remote server; it may dynamically load a remote JavaScript library and execute the code from the library (commonly used in web applications); it may also display an HTML-encoded text that comes from remote resources (JavaScript code may be embedded in the text). If these untrusted JavaScript programs are executed inside an application's WebView, there is no additional protection against the JavaScript code that comes from untrusted resource (e.g. remote web server) versus the code that comes from the developers. In all these scenarios, the JavaScript code from these resources may have the same privileges as the one installed by the users, unless the underlying access control mechanism can tell them apart and give them different privileges. Unfortunately, the PhoneGap framework in Android and iOS does not achieve this.

We did an study on the PhoneGap framework for Android. In the application that we used in this study, there are two regions: one is called the main frame, and the other is called subframe (achieved using `iframe`). The main frame contains the code from the developers and the subframe contains the code from a third-party. The application is given a set of permissions, which are meant to be given to the code running in the main frame. However, our investigation shows that the JavaScript code inside the subframe can use exactly the same permissions. For example, if the application is given a permission to send SMS messages, and if the application happens to load a third-party web page (e.g. advertisements) into its iframe, the code from the third party can also send SMS messages. Neither PhoneGap nor Android has an appropriate mechanism to protect against this kind of attacks.

Because HTML5-based applications use web technologies, they are potentially subject to most of the attacks that we have seen in the Web. Moreover, due to the holes on the sandbox, the damage of these attacks may be amplified. Imagine that an HTML5-based application is used to display SMS messages. Even though the app and the pages are benign, attackers can exploit vulnerabilities in the app by sending a malicious SMS message to the user, which contains JavaScript code. When the application reads the message and displays it to the user, the JavaScript code is executed. When that happens, the malicious JavaScript will have the same privileges as the application does, and can therefore read all the SMS messages on the phone, then send them to a remote server, and launch other attacks. The root cause of this attack is because the app may display messages using typical DOM APIs, such as `document.write`. If developers do not filter out the embedded JavaScript code, the JavaScript code will be invoked when the SMS message is displayed inside the web container. This is very much like the cross-site scripting (XSS) attack [33], which is still one of the most common attacks on web applications. If we do not provide a good access control for HTML5-based applications, this attack will soon find a new venue in mobile platforms, and so will other types of attacks on the Web. Being able to run HTML5-base mobile applications potentially amplifies the problems that we are still battling with in the Web. Although these attacks seem hypothetical, we have demonstrated that the XSS attack works. When more and more HTML5-based mobile applications are developed, attacks like this may soon become popular.

Contribution of this paper. In this paper, we make the following contributions: (1) We systematically study the "bridges" that expose mobile resources to JavaScript in order to support HTML5-based mobile apps. Based on the emerging threats that we have identified, we point out that most mobile systems' access controls support for HTML5-based mobile applications is not sufficient, because the assumptions that are true for native mobile applications may not be true anymore for HTML5-based applications. (2) We propose a fine-grained access control model for Android, which can provide a solid trusted computing base for HTML5-based applications. We have implemented it in Android.

In the rest of this paper, we first conduct an in-depth analysis of the access control for HTML5-based applications (Section 2). Then we present the design of a fine-grained access control system (Section 3). We have implemented our design in Android, which involves the modification of the operating system code and the WebKit engine [23]. We use a comprehensive evaluation to demonstrate the effectiveness and the efficiency of our design (Section 5).

2. THE PROBLEM

To support HTML5-based mobile applications, bridges need to be provided, so JavaScript code inside web container can access the system resources, which were originally blocked by the sandbox of web container. There are two typical approaches. One is used by Firefox OS and Tizen OS, which natively supports HTML5-based applications; we call it the *Native-API approach*. The second approach is used by the PhoneGap middleware and alike in Android and iOS; we call it the *Middleware approach*.

2.1 Native-API Approach

The most straightforward approach to support HTML5-based mobile applications is to directly implement system-access APIs inside web containers, and expose these APIs to the JavaScript engine, so JavaScript code can directly invoke these APIs and access system resources. Firefox OS and Tizen both use this approach.

Firefox OS uses Gecko [8] as application runtime to support mobile web apps; it provides a list of APIs [22] to allow JavaScript code to access system resources. At the same time, Gecko enforces runtime access control: whenever a Web API is requested, the access control module checks whether the app has the privilege to take the action or not (shown in Figure 1). It relies on the Same-Origin Policy to enforce app-level sandbox and it grants privileges based on the type of the app. A remotely loaded app can only request two Web APIs: geolocation and desktop notification; only privileged and certified applications are able to obtain more Web APIs, but they have to be local content or system apps. Additionally, when requesting certain critical Web APIs, such as geolocation, contacts, etc, the request will be prompted to users at runtime. Such kind of privileges are granted at the per-origin basis. The access control model used is called API-based access control.

2.2 Middleware Approach

Web content within the WebView component in Android or iOS does not have the direct API accesses to system resources. For the sake of clarity, we only use Android as an example in the following description. In Android, applications written in Java can access system APIs, because the APIs are already exposed to the Java Virtual Machine (Dalvik VM) either through Java classes or through JNI (Java Native Interface) [34]. These APIs are not exposed to the JavaScript engine. To allow JavaScript to access these APIs, a bridge is needed, which is written in Java and exposed to JavaScript. PhoneGap basically serves as this bridge.

PhoneGap is written in Java (and in Objective C for iOS). Applications that use the PhoneGap middleware need to include PhoneGap as a Java library. PhoneGap uses some techniques (discussed next) to allow the JavaScript code inside web container to invoke the PhoneGap APIs. Once PhoneGap code is invoked, it can further invoke system APIs on behalf of the JavaScript code. There exist two approaches that help JavaScript code invoke PhoneGap APIs that are written in Java: one is indirect invocation, and the other is direct invocation.

The Indirect Approach: Event-base Invocation. WebView allows Java code outside the web container to register an event listener, so when an event is generated inside the web container by JavaScript, the correponding event-listener code written in Java is triggered. In some sense, JavaScript code can indirectly invoke Java code. In Android, JavaScript code can generate a `Prompt` event, which triggers the `onJsPrompt` handler. Both iOS and Windows Phone have similar mechanisms. This is how the PhoneGap middleware implements the bridge between JavaScript and the system native APIs.

Direct Approach: Object-based Invocation. WebView in Android has another mechanism for JavaScript code to invoke Java code, and this mechanism is supported only in Android, not in iOS or Windows Phone. The goal of this approach is to allow applications to expose a Java object to the JavaScript engine, so JavaScript can directly invoke the APIs of this object, rather than going through the event-handler approach (see Figure 1). This is made possible by the `addJavascriptInterface` API in WebView. Applications can use this API to bind Java objects to WebView, so all the public methods of these Java objects can be directly invoked by the JavaScript code inside WebView.

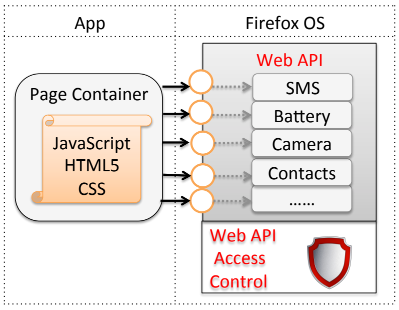
Functionality-wise, the indirect and direct approaches are similar; they are also similar to the two typical ways of how system calls are triggered in operating systems [29]. The indirect approach is similar to the “trapping to the kernel” approach using interruptions, and the direct approach is similar to the `SYSENTER/SYSEXIT` approach, which allows programs to directly enter the kernel space (only to dedicated entry points) without causing an interruption.

Performance-wise, the direct approach is more efficient than the indirect approach. That is why the `SYSENTER/SYSEXIT` instructions were created. That is also why PhoneGap switched from the first approach to the second one in its new version, after having been bought by Adobe. The switch was mostly motivated by the performance [1]. We also did a comparison, and the performance difference is significant, exceeding 30%, and can be more when scenarios get more complicated.

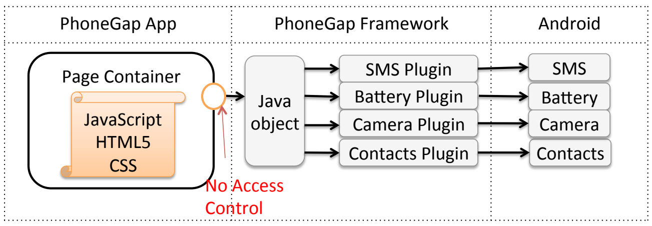
2.3 Security Problems

Android does not provide a system-level access control to protect the invocation from JavaScript to Java, whether the invocation is indirect or direct. It is application’s or middleware’s responsibility to enforce the access control. For the indirect approach, the job is easier, as there is a single entry point, i.e., the event handler; this place can be used for access control. Unfortunately, for the direct approach using `addJavascriptInterface`, implementing access control inside applications and middleware is problematic due to the following reasons.

First, when applications attach a Java object to WebView, it can perform access control based on the origin of the page inside WebView (typically using a whitelist, like what PhoneGap does), but once the object is attached, there is no more access control when the object’s APIs are invoked, unlike the `onJsPrompt()` approach, which can conduct access control on every invocation. This is not a problem if the contents inside WebView all come from the same origin. Unfortunately, when a Java object is attached to WebView, it is attached to all the frames inside WebView, such as iframes. These frames are not subject to the whitelist checking at the attaching time (only the main frame is subject to that access control); namely, even if their URLs are not on the whitelist,



(a) Native-API approach: Firefox OS



(b) Middleware approach: PhoneGap on Android

Figure 1: The native-API approach and middleware approach

they can still be loaded into iframes. The assumption behind this decision is that if we trust a web page, we should trust all the frames it contains; this is a wrong assumption.

Second, once a Java object is attached to WebView, there is no further restriction on what permissions an invocation can have. Currently, the invocation has all the application's permissions, even though it is potentially triggered by untrusted code. Clearly, there is a lack of granularity. There is no easy way for applications to restrict what permissions an invocation can have, because the permissions used by an invocation is not clear during the invocation time. For example, when a Java API is invoked by JavaScript, it is not easy to know whether the API will lead to the use of camera or not.

2.4 Our Approach

We believe that the current access control systems for Android and PhoneGap are not appropriate for supporting HTML5-based mobile applications. If the situation is not improved, the problem will get worse, because more and more developers are going to switch to developing HTML5-based applications. This calls for a research to study what kind of access control system is adequate for this emerging type of mobile applications. The solution should have the following properties:

- The solution should be built into Android, not PhoneGap. This is because PhoneGap is not the only framework that supports HTML5-based mobile application development. There are several other frameworks, including RhoMobile [18], Appcelerator [2], MoSync [14], WidgetPad [24], etc. A solution at the OS level can benefit all these frameworks.

- We should not give all the application's permissions to every frame inside WebView. Because the page in WebView can embed pages from different origins, it is important for the system to distinguish between the accesses initiated by untrusted origins from those by trusted ones, and give them different permissions.

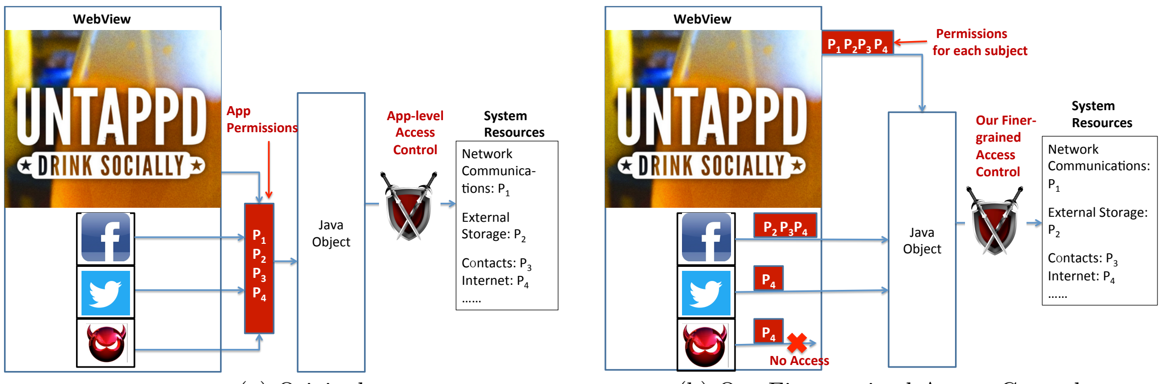
3. DESIGN

3.1 Access Control Model

Before talking about our access control model, we need to answer one question: why not use the API-based access control that is used by Firefox OS? Namely, the system can decide which APIs can be invoked by the JavaScript code inside WebView. There are three reasons that make Firefox's API-based model inappropriate for Android. First, as Felt et al. points out, in Android, there are more than 1000 APIs that involve privileged operations [32]. If we conduct API-based access control, developers need to decide which of these APIs can or cannot be invoked by the code inside WebView. This is simply impractical. Second, in Firefox OS, all the APIs exposed to JavaScript are system APIs, and their functionalities are well defined. However, in the WebView case, the APIs invoked by JavaScript code, whether through indirect invocation or direct invocation, are provided by applications or PhoneGap libraries; Android has no idea what these APIs are capable of doing. Relying on developers to tell the system exactly what privileged operations can be invoked by the APIs is not a reliable solution. Third, Android uses permission-based access control, so if we use API-based access control for WebView, these two models may not work well. For these reasons, we decide to adopt the same permission-based access control model in our work.

The fundamental problem of WebView is that when JavaScript code invokes Java code through the bridge on the web container, there is no isolation of privileges, so all the invocations from the bridge have the same privileges. In Android, this means all the invocations have the same permissions that are assigned to the application. A good access control system should be able to grant different permissions to different invocations, depending on where the invocations are initiated. Inside WebView, contents from multiple origins can co-exist, because each page inside WebView can have frames (such as iframes), which can contain pages from different URLs, some are more trustworthy than the others. Thus, different frames should be granted with different permissions.

We propose an access control model that allows developers to assign different permissions to different frames. We use an example to illustrate our model. Figure 2(a) illustrates the original access control model in Android, and our model is illustrated in Figure 2(b). In both figures, The main frame is called "UNTAPPD", and it has three iframe pages: Facebook, Twitter, and a page from an untrusted origin. The application has four permissions, $P_1, \ldots, P_4$. In Figure 2(a), we can see that all the iframes have these four permissions. In Figure 2(b), through the configuration provided by the developer, different sets of permissions are given to different frames, based on the requirement of the web pages and



(a) Original (b) Our Finer-grained Access Control

Figure 2: Access control in the original Android and our proposed model

how much developers trust them. We can even prohibit an untrusted page from invoking any Java API from its frame.

3.2 Policy Configuration

We provide two ways for developers to assign permissions. First, developers can assign permissions directly to each frame (in the HTML file), and this kind of permissions is called *frame permissions* (denoted by P_{frame}). Second, developers can also assign permissions to origins (in the manifest file), and these permissions are called *origin permissions* (denoted by P_{origin}). We also use P_{app} to represent the permission assigned to the entire application. The effective permissions ($P_{effective}$) for each frame is the intersection of these three types of permissions:

$$P_{effective} = P_{app} \bigcap P_{frame} \bigcap P_{origin} \qquad (1)$$

It should be noted that P_{origin} can change, so every time the origin inside a frame changes, a new $P_{effective}$ will be calculated for this frame. The definition of "origin" in our model is the same as the "origin" in the Same-Origin Policy, i.e., it is the unique combinations of three elements: port, scheme, and domain. Because the `INTERNET` permission is assumed to be a necessity for mobile web applications, developers do not have to set this permission for frames or origins as long as it is requested for the application.

Frame Permission Assignment. Developers can specify frame permissions using the `permissions` attribute for frames (see Figure 3) in HTML code. The field can be either empty, NULL, or contain a list of permission names separated by space. If this attribute is not specified, it automatically inherits the permissions of its parent frames. The NULL permission means that the JavaScript code inside this frame cannot invoke any Java API: there is no bridge for this frame. This is different from an empty permission list: an empty list means that JavaScript code inside the frame can invoke Java APIs, but the execution of the Java code does not have any Android permission.

Conditions	Code
Non-empty permissions	<iframe permissions ="READ_CONTACTS" src = "http://www.facebook.com"/>
Empty permission	<iframe permissions =" " src = "http://www.facebook.com"/>
NULL	<iframe permissions ="NULL" src = "http://www.facebook.com"/>
No frame policy	<iframe src = "http://www.facebook.com"/>

Figure 3: Frame Permission Configuration

Origin Permission Assignment. We use an example to explain how to assign origin permissions. As shown in Figure 2, the "Untappd" app requires the following permissions: `INTERNET`, `ACCESS_NETWORK_STATE`, `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`, `READ_CONTACT`, etc. Developers configure origin permissions in the manifest file. We introduce a new tag called "access", which uses an attribute called "origin" to specify origins; we also introduce a nested tag called "origin-permission", which specifies the permissions assigned to the origin. See the following example.

```
<access origin=http://*.untappd.com>
     <origin-permission android:name="ACCESS_NETWORK_STATE" />
     <origin-permission android:name="READ_EXTERNAL_STORAGE" />
     <origin-permission android:name="WRITE_EXTERNAL_STORAGE" />
     <origin-permission android:name="READ_CONTACT" />
</access>
<access origin=http://*.facebook.com>
     <origin-permission android:name="READ_CONTACT" />
     <origin-permission android:name="READ_EXTERNAL_STORAGE" />
     <origin-permission android:name="WRITE_EXTERNAL_STORAGE" />
</access>
<access origin=http://*.twitter.com></access>
```

Note: JavaScript code in WebView can access system resources through two types of APIs: one is the native APIs exposed by the WebView to the pages inside, such as DOM APIs. The other type is the application APIs (written in Java) exposed to WebView through `addJavascriptInterface()` and the event-handler mechanism. We focus on the second type. Our solution can be easily extended to cover the first type.

Policy Enforcement. To enforce the security policy described above, we need to solve the following three problems: (1) Where to store the policy information in the system? (2) How to intercept the invocation from JavaScript code to native Java code through the bridge, so we can set the effective permissions for this invocation? (3) Where to check the effective permissions in order to enforce access control? We discuss our design decisions based on these three problems.

3.3 Assigning Effective Permission to Frame

As we have discussed above, our security policies are specified in two places: in HTML pages (for frame permissions) and in the manifest file (for origin permissions). To read the permission information, we extend the existing manifest file parser and the HTML parser (in Webkit) to retrieve the permission information from these two places. The origin permissions will be stored in the same place where Android stores permissions for each application. The frame permission is stored as an attribute of the frame object in the DOM tree. Using Equation (1), we can calculate the effective permission of a frame.

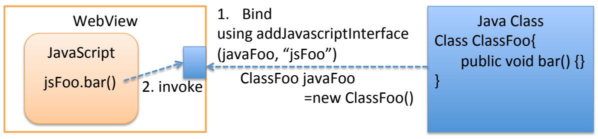


Figure 4: How JavaScript invokes a Java API

Every time a page is loaded into a frame, an object called `SecurityOrigin` is created, and this object is used by browsers to enforce the same-origin policy, so JavaScript code from one origin cannot access the resources belonging to other origins. There is one `SecurityOrigin` object per frame. We store the frame's effective permission in this object. Every time a new page is loaded into a frame, the effective permissions will be recalculated, and the corresponding values in `SecurityOrigin` will be updated accordingly.

3.4 Setting Effective Permission at Invocation

Setting the effective permissions for each frame is not enough, we need to ensure that when JavaScript inside a frame invokes a Java API through the bridge, the API is executed only with the frame's effective permissions, not the application's permissions. To achieve this goal, we need to understand how JavaScript invokes a Java API in Android; we need to intercept this invocation, and set the effective permissions before the API code starts running.

As we have discussed before, there are two ways for JavaScript in WebView to invoke Java code: indirect invocation and direct invocation. The indirect invocation is quite straightforward, and easy to be intercepted. The direct invocation, i.e., through the APIs attached by `addJavascriptInterface()`, is quite complicated. In the following, we only focus on the direct invocation method.

How JavaScript invokes Java code. In browsers, it is often necessary to allow JavaScript to interact with browser plugins, such as Flash, PDF reader, Java Applet, etc. A de facto standard for such an interaction was initially developed for Netscape, but was subsequently implemented by many other browsers [15]. It is called Netscape Plugin Application Programming Interface (NPAPI) [15], which provides a cross-platform plugin architecture for browsers. It allows JavaScript within a browser to access the APIs of plug-ins, and vice versa. In Android, from the WebView's perspective, Java is treated just as a plugin, and invocation of Java code from JavaScript follows the NPAPI standard.

To explain how NPAPI works, we use an example, which is depicted in Figure 4. In this example, there is a Java class called `ClassFoo`, and an instance of this class called `javaFoo`. This instance is bound to WebView through `addJavascriptInterface()`, resulting in a new JavaScript object called `jsFoo` in WebView. When the JavaScript code in WebView invokes `jsFoo.bar()`, a series of actions will be performed, leading to the eventual invocation of the `bar` method of the Java object `javaFoo`.

Figure 5 shows how the APIs of plugins (Java object, C# object, etc) are provided to JavaScript. We will use Java as an example to illustrate the process. First, we need to make the APIs of the Java object `javaFoo` invokable by the code outside Java Virtual Machine (JVM). This is done through JNI: the Java object `javaFoo` is converted into a JNI object, which is further wrapped into a new object called `JavaInstance`, along with the JVM environment, so native code

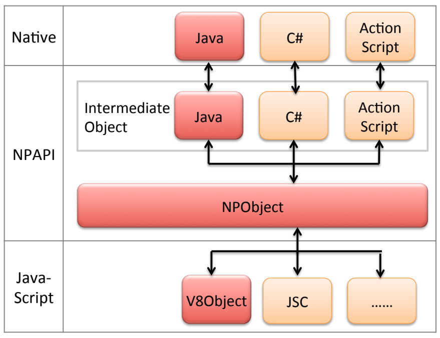


Figure 5: The NPAPI Architecture

(written in C++) can call the methods of this Java object and access its fields. `JavaInstance` is further wrapped into an object called `NPObject`, which is defined in the NPAPI architecture. Finally, a JavaScript object called `V8Object` is created (Android uses the V8 JavaScript engine), and this object contains a reference to the `NPObject` and essential callback functions. This object is then given a name `jsFoo` (specified by `addJavascriptInterface()`) and made available to JavaScript code. After this is done, JavaScript code can invoke `jsFoo.bar()`, which will eventually lead to the invocation of `javaFoo.bar()`.

Invocation reverses the above wrapping process. When `V8Object` is created, a callback function (defined in `NPObject`) is registered to this JavaScript object, so when any method is invoked, the callback function will be invoked. Therefore, when JavaScript calls `jsFoo.bar()`, the callback function is triggered; it retrieves the `NPObject` object, extracts the `JavaInstance` object, and eventually gets the JNI object. After that, the Java function is called through the JNI mechanism: basically, it invokes the attached JVM instance, passes to it the Java object reference, method name, and arguments, and JVM does the rest.

Adjusting the Effective Permission at Runtime. The current NPAPI design in Android does not change the effective permissions when JavaScript invokes Java; that is why the invocation automatically inherits all the application's permissions. To change that, we need to intercept the invocation, identify which frame the invocation comes from, retrieve the effective permissions stored in the frame's `SecurityOrigin`, and then set the effective permissions of the invocation. From Figure 5, we can see that there are several places where we can intercept the invocation. We choose the place between `V8Object` and `NPObject`, because at this point, it is easier to get the frame information. Moreover, this design does not depend on the types of plugins, so it also works if the plugins are not Java, but C# or Flash, etc.

Once we intercept the invocation, we need to set the effective permissions of the runtime environment, so the system knows what permissions to check when the code triggered by this invocation tries to access protected resources. The question is whether we should do it at the process level or thread level. In Android, the invocation from JavaScript to Java is thread-based, so the effective permissions can only be set at the thread level, not the process level. Android is built upon Linux, and in the Linux kernel, threads are represented as `task_struct`. Whenever an application creates a new user-space thread, a new `task_struct` is created in the kernel. We attach the effective permissions to this thread data structure.

Although the API invocation runs in a single thread in WebKit, an application may create another thread to conduct the task. This is the case in PhoneGap: when a PhoneGap API is invoked, one thread is used, but PhoneGap quickly spawns another thread and hands the rest of the job to this new thread, so it can return to accept another invocation. We need to ensure that the effective permissions are also handed over to the new thread, so it has adequate permissions to finish the task. What makes things more complicated is that in PhoneGap and many other applications, the new thread is not actually created, but fetched from the idle pool, i.e., the thread is already created. We need to set the effective permissions correctly, and clear the effective permissions before returning the thread back to the idle pool. We use the hook functions of thread to set and clear the permissions accordingly.

3.5 Checking Effective Permissions

We extend Android's existing *Reference Monitor* to check the effective permissions when an application tries to access protected resources, such as external storage, camera, contact, etc. In Android's original reference monitor, the application's User ID (UID) is used to find out the permissions of the application, and access control is conducted based on these permissions. To enforce our access control model, we need to use Thread ID (TID) to find out the effective permissions of the current thread, and then conduct access control based on the effective permissions. This only involves small changes to the existing reference monitor. For backward compatibility, if a thread does not have an effective permission list, access control will fall back to the original Android access control model, so we will not break the existing applications. However, we ensure that every thread involved in the JavaScript-to-Java invocation will have an effective permission list, even if the list is empty.

4. POTENTIAL ATTACKS

There can be some potential attacks against our access control system. We discuss how our system handles these potential attacks.

A Principal Increasing Privilege. A JavaScript program may attempt to get more privileges using the DOM API function `setAttribute` to modify the permission attribute of the iframe tag [25]. However, the configuration information is not exposed to JavaScript program. Therefore, such attempts to modify the attributes cannot succeed.

One page can use `document.domain` to change its origin to a superdomain [50]. If developers give `b.com` more permissions than its sub-domain `a.b.com`, a page in `a.b.com` can change its origin to `b.com`, and can thus gain more privileges. Our design will track this change, check the permissions assigned to the previous frame and calculate the conjunction of both permission sets, and use the result as the effective permissions.

Scoping Rule. JavaScript programs from an iframe can create nested iframes. A malicious JavaScript program may attempt to use this feature to create an iframe with higher privileges than the parent frame. Therefore, our design enforces a **scoping rule** to protect against such attempts. The scoping rule ensures that the permissions assigned to child frames cannot exceed the main frame's own permissions. Formally speaking, when a frame tag is labeled with permission="p1, p2", then the privileges of the principals within the scope of this frame, including all sub frames, are bounded by {p1, p2}. Our framework implementation strictly enforces this rule. Our scoping rule also applies to the pop-up windows.

Changing Other Frame's Location. JavaScript programs in one frame can change the location of the main frame page even if they come from different origins [36]. In a potential attack, an untrusted page is restricted to an iframe with very few permissions. To gain more power, the JavaScript program in this page can navigate the main frame to the page's URL, so this untrusted page is loaded into the main frame. Although the page is still subject to the origin-based security policy, the frame-based policy is successfully negated. Our design enforces a strict policy to prevent the script from lower privileged frame to make any change to the frames of higher privilege.

5. EVALUATION

In this section, we evaluate our work on two aspects: effectiveness and performance. First, we use different cases to show how our work can be effectively used to isolate the privilege of web components at the frame level. Then, we evaluate the overhead caused by our work.

5.1 Restrict Permissions of Untrusted Frames

In this experiment, we show how our work can achieve privilege isolation for frames. In all our experiments, we assume that the `INTERNET` permission is granted. We use a PhoneGap application called HealthTap [9], which is a very popular app with more than one million downloads. We added several contact entries into the phone, one of which is Alice, with phone number and email address. We will use this information in our demonstration.

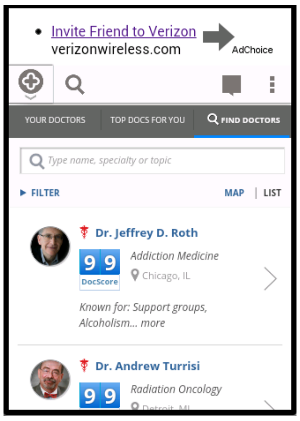
HealthTap requires the following permissions: `SEND_SMS`, `READ_EXTERNAL_STORAGE`, `READ_PHONE_STATE`, `GET_ACCOUNTS`, `INTERNET`, `WAKE_LOCK`, `ACCESS_LOCATION_EXTRA_COMMANDS`, `ACCESS_COARSE_LOCATION`, `READ_CONTACTS`, `ACCESS_NETWORK _STATE`, `ACCESS_FINE_LOCATION`, `CAMERA`, `BROADCAST_STICKY`.

Several permissions are related to mobile system resources. Once granted, all the frames in this app can access the corresponding mobile system resources, using the app's full privilege, through the bridge attached by PhoneGap. With our work, we can easily limit the privileges of the frames without affecting the original application. We achieve that by setting the origin permissions and/or frame permissions.

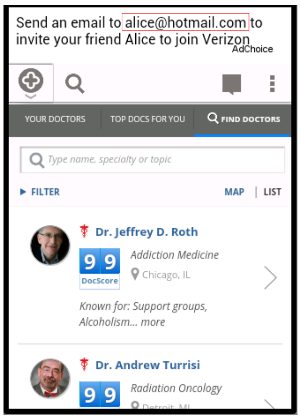
To show how our work can limit the privileges of frames, we need to slightly change the app. We first use Apktool to disassemble the application APK file; then we add an iframe into the original app's webpage; finally we repackage the APK file with the modified manifest file and sign it with our key. Now we can demonstrate how to limit the privileges of iframes.

Advertisement. The most common case of using iframe is to load advertisement. We construct a Verizon advertisement webpage, host it on a remote server, and load it in an iframe inside the modified app. The app is shown in Figure 6(a). This advertisement needs to invite more people to join Verizon, so it has to read the contacts on the phone and send invitations to friends. We specify the origin permissions for the advertisement's origin as the following:

```
<access origin="http://*.verizonads.com">
  <origin-permission
      android:name="android.permission.READ_CONTACTS"/>
</access>
```



(a) Screenshot of app


Send an email to alice@hotmail.com to invite your friend Alice to join Verizon
AdChoice
YOUR DOCTORS
TOP DOCS FOR YOU
FIND DOCTORS
Type name, specialty or topic
FILTER
MAP
LIST
Dr. Jeffrey D. Roth
Addiction Medicine
Chicago, IL
Known for: Support groups, Alcoholism... more
Dr. Andrew Turrisi
Radiation Oncology

(b) Invite friend to Verizon

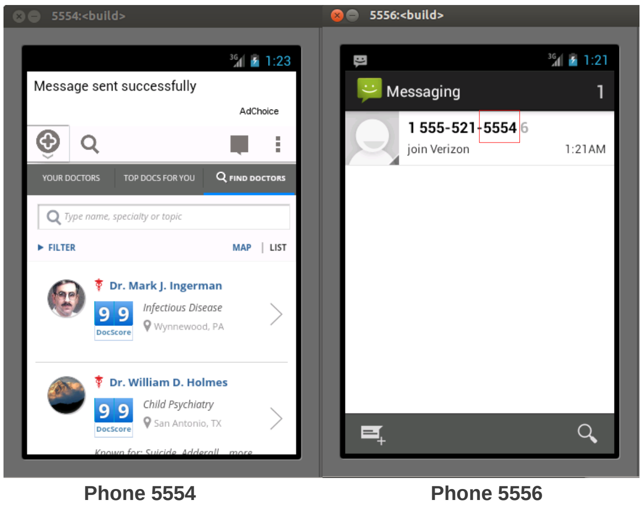
Figure 6: App with Advertisement

From our setup, the effective permission of the iframe is limited to `READ_CONTACTS`, i.e., the iframe can only read the contact, but cannot access other mobile resources such as SMS message. If the advertisement tries to collect more information from the user, it can only get the contact information and nothing else. Figure 6(b) shows that the page in the iframe gets Alice's contact information on the phone, but if it tries to send SMS messages to other phones, a security exception will be thrown (see Figure 7(b)) and the request will be denied, because this iframe does not have the `SEND_SMS` permission. In the original Android system, if the advertisement tries to send SMS messages, it will succeed (see Figure 7(a)), because the app does have the `SEND_SMS` permission.

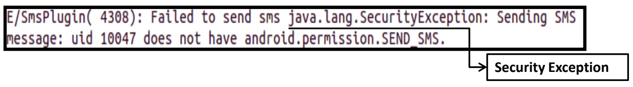
Social Network Plugin. Developers often include social network plugins, such as the Facebook "Like" button, in their applications to attract more users, and iframes is widely used to load social network plugins. In this evaluation, we load a Facebook "Like" button into an iframe of the `HealthTap` app (see Figure 8(a)). Since we know that these kinds of social network plugin do not need to require mobile system resources, we should not give this iframe any permission. We can give this frame an empty permission list:

```
<iframe src="http://www.facebook.com/plugins/like.php?href=https
%3A%2F%2Fwww.facebook.com%2FHealthTap&amp;&amp;" permissions="">
</iframe>
```

From Figure 8(b), we can see that in the mainframe we can still get Alice's email address. But if attackers try to get the contacts in the iframe, a "Permission Denial" exception will be thrown (see Figure 8(c)). This is because the frame permission of this iframe is empty, i.e., it can invoke Java APIs, but will not be able to have any permission during the execution. In this situation, we can see that although the app has the `READ_CONTACTS` permission, the iframe that loads the Facebook plugin does not have the permission.



(a) Phone5554 successfully sends an SMS to Phone5556



```
E/SmsPlugin( 4308): Failed to send sms java.lang.SecurityException: Sending SMS
message: uid 10047 does not have android.permission.SEND_SMS.
```

(b) Fail to send SMS

Figure 7: Send SMS

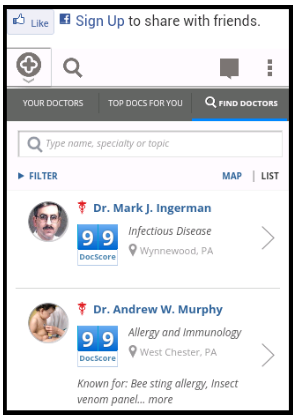
Sometimes, if the social network plugin is a faked one, not from Facebook, it will fail if it tries to invoke Java APIs. As Figure 8(d) shows, the API cannot be found. This is because the origin is not in the origin permissions list, which means this origin is not trusted, and the JavaScript code from this origin cannot invoke any Java API through the WebView bridges.

5.2 Reduce Damages Caused by Mistakes

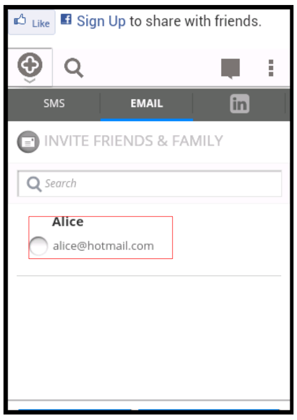
The cross-site scripting (XSS) attack is one of the most common attacks on web applications. We demonstrate that if mobile applications are written using the web technologies, XSS attacks can be launched against mobile applications as well. We then demonstrate how our access control can help reduce the damage.

For the demonstration purpose, we wrote an HTML5-based mobile app; one of the functionalities of this app is to display SMS messages. The app is given the `INTERNET`, `READ_CONTACTS`, and `READ_SMS` permissions. The app uses PhoneGap to access the system resources, such as SMS and contacts. To display SMS messages, the app invokes commonly-used APIs, including the DOM API `document.write` and JQuery APIs. It should be noted that if a message contains JavaScript code, these APIs will execute the code, instead of displaying it.

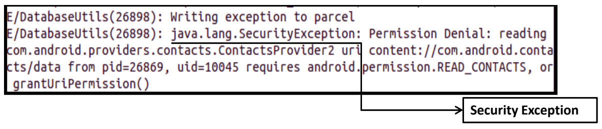
This mobile app is vulnerable to XSS attacks. Attackers can send to their victims an SMS message that contains malicious JavaScript code. In Figure 9, Phone 5554 sends to Phone 5556 such an SMS message. The app we wrote reads the SMS message and displays it. From the figure, we can see that the embedded JavaScript code is executed, and the victim's contact information is retrieved by this JavaScript code. The situation will be worse if the app has more permissions such as `WRITE_CONTACTS` and `SEND_SMS`, because the malicious JavaScript can then delete the victim's contacts



(a) Screenshot of app

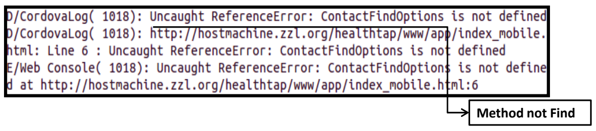


(b) Mainframe get contact



```
E/DatabaseUtils(26898): Writing exception to parcel
E/DatabaseUtils(26898): java.lang.SecurityException: Permission Denial: reading
com.android.providers.contacts.ContactsProvider2 uri content://com.android.conta
cts/data from pid=26869, uid=10045 requires android.permission.READ_CONTACTS, or
 grantUriPermission()
```

(c) Security exception caused by not having enough permission to read contact



```
D/CordovaLog( 1018): Uncaught ReferenceError: ContactFindOptions is not defined
D/CordovaLog( 1018): http://hostmachine.zzl.org/healthtap/www/app/index_mobile.
html: Line 6 : Uncaught ReferenceError: ContactFindOptions is not defined
E/Web Console( 1018): Uncaught ReferenceError: ContactFindOptions is not define
d at http://hostmachine.zzl.org/healthtap/www/app/index_mobile.html:6
```

(d) Cannot find Java object

Figure 8: App with Social Network Plugin

and send out SMS messages. We have installed this app on real phones (using Verizon and T-Mobile), and the attack also works. Apparently, these service providers do not filter out JavaScript code in SMS messages, as nobody has demonstrated such an attack before. Although the app we wrote is intentionally built for the demonstration purpose, the error in the code does represent a very typical error that many developers can make.

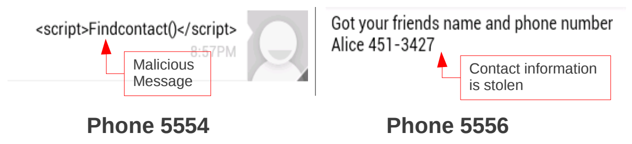

<script>Findcontact()</script>
8:57PM
Malicious Message
Phone 5554
Got your friends name and phone number
Alice 451-3427
Contact information is stolen
Phone 5556

Figure 9: Phone5554 sends a malicious SMS message to Phone5556

Our access control mechanism restricts application's privileges at the frame level, so it may not solve this problem completely; however, using our access control mechanism, we can place such an error-prone code inside an iframe, and give this iframe a very limited set of permissions. This way, even if the malicious JavaScript code in the SMS message is triggered, its damage will be limited. In our future work, we will systematically study this kind of attacks and develop finer-grained access control to deal with this new threat.

5.3 Performance Evaluation

Our environment is set up as the following: we use Jelly Bean (*android-4.2.1_r2*), and we run our modified Android in Unbuntu 11.04 using an emulator. The configuration for the emulator is the following: Nexus One (3.7, 480*720: hdpi), RAM (2 GB), VM Heap (32 MB), Internal Storage (200 MB), and No SDCard. The hosting machine is Intel Core i7-3540M @ 3.00GHZ, with 8GB memory.

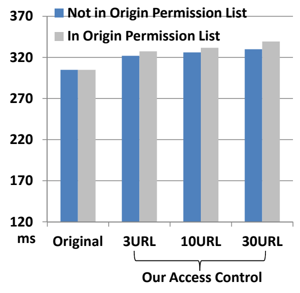
System Overhead. To evaluate the performance of our system, we use a popular Android Benchmarking tool: AnTuTu. It runs a set of tests and provides a score report about memory performance, CPU integer performance and CPU floating point performance etc. Table 1 shows the results (average scores of multiple runs) of the AnTuTu Benchmark. From the benchmark, we can observe that our modification imposes no significant overhead on these parameters: all overheads are below 1%.

AnTuTu	Original SDK	Modified SDK	Difference
Total Score	1254.2	1252.7	-0.12%
Memory Access	387.2	386.5	-0.18%
Integer Operation	454.6	456.2	0.35%
Float-point Operation	134.1	134	-0.01%
2-D Graphic	39.3	39	-0.77%
3-D Graphic	26.5	26.3	-0.75%
Database IO	212.5	210.7	-0.85%

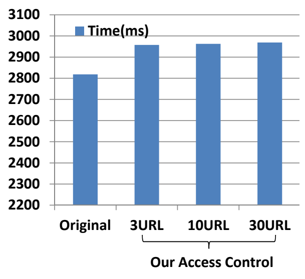
Table 1: Benchmark Scores

Application Overhead. We evaluate the performance impact of our work on applications. We mainly focus on the applications that attach APIs to WebView using `addJavascriptInterface()`. We measure the followings: overhead in page loading and overhead in API invocations. They are the places where we made the most changes in Android.

Because the need to calculate the effective permissions when a page is loaded, there will be overhead in page loading. The overhead is shown in Figure 10(a). If the webpage is not in the origin-permission list, the overhead is about 5.6% to 8.2%, depending on the number of URLs in the list. If the webpage is in the list, the overhead is a little bit higher: about 7.4% to 11.3%. This is only a one-time cost.



(a) Loading Overhead



(b) Invocation Overhead

Figure 10: Overhead

When JavaScript code calls the developer-specified APIs from the HTML code, permissions are checked either in the kernel or at the framework. Figure 10(b) shows the result of how much time we need to invoke 100 APIs. As it shows, in our modified Android System the invocation time is about 4.9% higher than the original system when there are 3 URLs in the origin-permission list. The overhead is 5.3% when there are 30 URLs in the origin permissions list, which is almost the same as 3 URLs.

6. RELATED WORK

Privilege Separation and Isolation in Web A large number of works have been proposed to limit the privilege of JavaScript in web applications: The sandbox attribute of iframe [26] allows developers to decide whether to allow JavaScript to execute or not; it has been implemented by most browsers and is also very coarse-grained. Content Security Policy [4] and ConSCRIPT [45] present client-side fine-grained and application-specific security policies. Adjail [40] focuses on isolating the JavaScript of third-party advertisements by creating a shadow page and letting ads to run inside it. Maffeis et al. [42, 43] propose a language-based approach to filter and rewrite untrusted JavaScript. Similarly, Caja [3] and ADsafe [31] use a safe subset of JavaScript, and they eliminate dangerous DOM APIs such as `eval` and `document.write`, which could allow advertisements to take control of the entire webpage. Several works focus on sandbox components in mashups [46,52]. A representative work of the holistic approach is the Escudo work [37]: Escudo proposes a ring-based access control model for web browsers. Zhou and Evans [53] propose to isolate third-party JavaScript from accessing private user data within webpage. Akhawe et al. [28] propose a temporary origin-based approach to enforce privilege separation for HTML5 apps.

Privilege Separation and Isolation in Android Advertisements are critical third-party components of mobile applications, and they have the same privileges as the hosting app. Several works attempt to address this problem by separating advertisements' privileges from the apps: AdSplit [48] isolates the advertisement into a separate process so that ads will have a separate set of permissions; Leontiadis et al. [39] also use a separate application to host the advertisement with IPC to support communications with the app. AdDroid [47] provides its own advertisement SDK and ads-specific permissions aiming at protecting user privacy. Jeon et al. [38] propose a finer-grained access control by splitting existing permissions into multiple finer-grained ones to align with Least Privilege principle.

Security Concerns of WebView The problems with the `addJavascriptInterface` in WebView were initially identified by Luo et al. [41]. Another work mainly focuses on how the exposed JavaScript interface is used in ads library and their privacy concerns [51].

Expose Mobile Resources to Web Agarwal et al. [12] and Adappa et al. [27] propose a middleware design to expose mobile system resources to mobile mashups [44]. They define a fine-grained policy to specify a list of APIs that one component of Mashup can access. However, their approaches do not have frame-level policy like ours and their access control is enforced at the exposed API point. If developers write their own native Java objects and then expose them to JavaScript, their access control will be bypassed. Similarly, Singh proposes to extend the origin concept to mobile web applications and also propose a design to expose APIs to the origins of one web applications in Android [49], the architecture is similar to Firefox OS and the work discussed above. As we argued in Section 3, our system-level permission-based solution is more appropriate in Android than the API-based access control.

7. SUMMARY

In this paper, we study the potential security problems of the HTML5-based applications in mobile systems. We have identified the insufficiency of the access control in the existing platforms. We propose a fine-grained access control mechanism to support HTML5-based applications in Android platform. In our access control, we define a frame-based and origin-based policy to separate subjects within the same application. We enforce our access control in the operating system, so developers of the HTML5-based applications only need to configure their security policies, without worrying about implementing the enforcement by themselves. Our implementation only requires light-weight code modification of the original Android system, and poses only small overhead. The evaluation demonstrates that our access control prototype can effectively separate privileges for different principles within the same app.

8. REFERENCES

[1] `http://mail-archives.apache.org/mod_mbox/incubator-callback-dev/201208.mbox/\%3C184786566.604.1345647222886.JavaMail.jiratomcat\@arcas\%3E/`.

[2] Appcelerator platform. `http://www.appcelerator.com/`.

[3] Caja. `http://code.google.com/p/google-caja/`.

[4] Content security policy. `http://www.w3.org/TR/CSP/`.

[5] Facebook social plugin. `https://developers.facebook.com/docs/plugins/`.

[6] Firefox os. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS`.

[7] The future of mobile development: Html5 vs. native apps. `http://www.businessinsider.com/html5-vs-native-apps-for-mobile-2013-4?op=1/`.

[8] Gecko. `https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/`.

[9] Healthtap application. `https://www.healthtap.com/`.

[10] Html5 vs. apps: Where the debate stands now, and why it matters. `http://www.businessinsider.com/html5-vs-apps-where-the-debate-stands-now-and\-why-it-matters-2013-4/`.

[11] Html5 vs native: The mobile app debate. `http://www.html5rocks.com/en/mobile/nativedebate/`.

[12] A middleware framework for mashing device and telecom features with the web. `http://domino.research.ibm.com/library/cyberdig.nsf/papers/78674C5DF95D79D08525776F0045E601/`.

[13] Mobile app. `http://en.wikipedia.org/wiki/Mobile_app`.

[14] Mosync: App development made easy. `http://www.mosync.com//`.

[15] Npapi. `http://en.wikipedia.org/wiki/NPAPI/`.

[16] Phonegap best and free cross-platform mobile app framework. `http://crossplatformappmart.blogspot.com/2013/03/phonegap-best-free-cross-platform.html`.

[17] Phonegap: Easily create apps using the web technologies you know and love: Html, css and javascript. http://phonegap.com.

[18] Rhomobile suite. http://www.motorolasolutions.com/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite.

[19] The shared future of html5 and native apps. http://www.itbusinessedge.com/blogs/data-and-telecom/the-shared-future-of-html5-and-native-apps.html/.

[20] Tizen. https://www.tizen.org/.

[21] Twitter social plugin. http://www.addthis.com/social-plugins/tweet-button/.

[22] Webapi. https://wiki.mozilla.org/WebAPI/.

[23] The webkit open source project. http://www.webkit.org/.

[24] Widgetpad: Open-source, web-based environment for mobile developers. http://readwrite.com/2009/09/21/widgetpad/.

[25] Xml dom setattribute() method. http://www.w3schools.com/dom/met_element_setattribute.asp/.

[26] Html5 sandbox attribute. http://www.whatwg.org/specs/web-apps/current-work/#attr-iframe-sandbox, 2010.

[27] S. Adappa, V. Agarwal, S. Goyal, P. Kumaraguru, and S. Mittal. User controllable security and privacy for mobile mashups. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 35–40, New York, NY, USA, 2011. ACM.

[28] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association.

[29] D. Bovet and M. Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[30] A. Charland and B. Leroux. Mobile application development: web vs. native. *Commun. ACM*, 54(5):49–53, May 2011.

[31] D. Crockford. ADSafe. http://www.adsafe.org.

[32] 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*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[33] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense.* Syngress Publishing, 2007.

[34] R. Gordon. *Essential JNI: Java Native Interface.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[35] N. Huy and D. vanThanh. Evaluation of mobile app paradigms. In *Proceedings of the 10th International Conference on Advances in Mobile Computing &#38; Multimedia*, MoMM '12, pages 25–30, New York, NY, USA, 2012. ACM.

[36] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 611–620, New York, NY, USA, 2007. ACM.

[37] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 231–240, Washington, DC, USA, 2010. IEEE Computer Society.

[38] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[39] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems &#38; Applications*, HotMobile '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[40] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[41] T. Luo, Hao Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 343–352, New York, NY, USA, 2011. ACM.

[42] S. Maffeis, J. C. Mitchell, and A. Taly. Isolating javascript with filters, rewriting, and wrappers. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 505–522, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] S. Maffeis and A. Taly. Language-based isolation of untrusted javascript. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 77–91, Washington, DC, USA, 2009. IEEE Computer Society.

[44] E. M. Maximilien. Mobile mashups: Thoughts, directions, and challenges. In *Semantic Computing, 2008 IEEE International Conference on*, pages 597–600, 2008.

[45] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 481–496, Washington, DC, USA, 2010. IEEE Computer Society.

[46] J. Mickens and M. Finifter. Jigsaw: efficient, low-effort mashup isolation. In *Proceedings of the 3rd USENIX conference on Web Application Development*, WebApps'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[47] P. Paul, P. F. Adrienne, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and*

Communications Security, AsiaCCS '12, 2012.

[48] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX conference on Security symposium*, USENIX Security '12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[49] K. Singh. Can Mobile learn from the Web? In *IEEE Computer Society Security and Privacy Workshops*, WPSP '12, 2012.

[50] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 463–478, Washington, DC, USA, 2010. IEEE Computer Society.

[51] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating User Privacy in Android Ad Libraries. In *IEEE Mobile Security Technologies (MoST) 2012*, MoST '12, 2012.

[52] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 1–16, New York, NY, USA, 2007. ACM.

[53] Y. Zhou and D. Evans. Protecting private web content from embedded scripts. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 60–79, Berlin, Heidelberg, 2011. Springer-Verlag.