Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks

Martin Georgiev
The University of Texas at Austin
mgeorgiev@utexas.edu

Suman Jana
The University of Texas at Austin
suman@cs.utexas.edu

Vitaly Shmatikov
The University of Texas at Austin

***Abstract*—Hybrid mobile applications (apps) combine the features of Web applications and "native" mobile apps. Like Web applications, they are implemented in portable, platform-independent languages such as HTML and JavaScript. Like native apps, they have direct access to local device resources—file system, location, camera, contacts, etc.**

Hybrid apps are typically developed using hybrid application frameworks such as PhoneGap. The purpose of the framework is twofold. First, it provides an embedded Web browser (for example, WebView on Android) that executes the app's Web code. Second, it supplies "bridges" that allow Web code to escape the browser and access local resources on the device.

We analyze the software stack created by hybrid frameworks and demonstrate that it does not properly compose the access-control policies governing Web code and local code, respectively. Web code is governed by the same origin policy, whereas local code is governed by the access-control policy of the operating system (for example, user-granted permissions in Android). The bridges added by the framework to the browser have the same local access rights as the entire application, but are not correctly protected by the same origin policy. This opens the door to *fracking* attacks, which allow foreign-origin Web content included into a hybrid app (e.g., ads confined in iframes) to drill through the layers and directly access device resources. Fracking vulnerabilities are generic: they affect all hybrid frameworks, all embedded Web browsers, all bridge mechanisms, and all platforms on which these frameworks are deployed.

We study the prevalence of fracking vulnerabilities in free Android apps based on the PhoneGap framework. Each vulnerability exposes sensitive local resources—the ability to read and write contacts list, local files, etc.—to dozens of potentially malicious Web domains. We also analyze the defenses deployed by hybrid frameworks to prevent resource access by foreign-origin Web content and explain why they are ineffectual.

We then present NoFrak, a capability-based defense against fracking attacks. NoFrak is platform-independent, compatible with any framework and embedded browser, requires no changes to the code of the existing hybrid apps, and does not break their advertising-supported business model.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.

NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23323

I. Introduction

Web apps are becoming more like native mobile apps, and vice versa. When running on mobile devices, modern Web apps often need to break out of the browser sandbox and directly access local resources such as the camera, geolocation, file system, etc. At the same time, many developers of mobile apps prefer to write them in platform-independent, implement-once-run-everywhere Web languages like JavaScript. The resulting "hybrid" apps thus exhibit features of both Web and native apps.

Web browsers are beginning to add mechanisms that expose local resources to Web code, but currently most hybrid apps are developed using *hybrid application frameworks* such as PhoneGap. The primary purpose of these frameworks is to supply *bridges* that provide Web code with direct access to local resources on the machine. These frameworks thus support the development of portable mobile apps and the conversion of existing Web apps into mobile apps. Their target platforms include mobile-phone operating systems (OS) such as Android, iOS, Windows Phone, and BlackBerry, as well as desktop OSes such as MacOS.

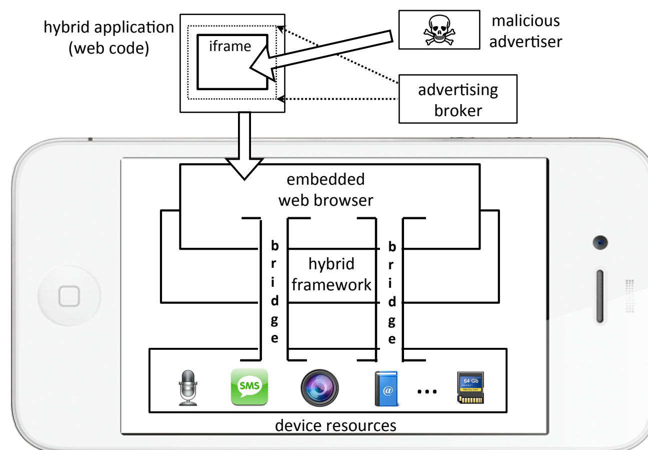

Fig. 1: Hybrid software stack

The software stack created by hybrid frameworks is sketched in Fig. 1. A hybrid framework comprises two halves. The local half is implemented in a platform-specific language like Java, C#, or Objective-C. It runs as a local process in the host machine's OS and performs actual device access: reading and writing the file system, working with the camera, etc. To execute the app, the local half creates an instance of a platform-specific embedded Web browser—for example, WebView in

Android, UIWebView in iOS, or WebBrowser in Windows Phone—and runs the app's code within this browser. The Web half of the framework is a JavaScript library. The app's code includes this JavaScript library and uses its API to access local resources on the device.

The access-control model of the software stack shown in Fig. 1 is quite complex. Web content is governed by the same origin policy, while local resource accesses are governed by the OS's discretionary access control policy. In contrast to advertising-supported native apps, untrusted content in hybrid apps is not included via a separate local library. Instead, it is composed with the app's own content like a Web mashup, with different origins isolated in separate iframes, all executing together in the same Web browser. Since the hybrid framework provides Web content with the ability to access local resources, it must correctly propagate origin information and *apply origin-based access control to resources outside the Web browser*. In particular, it must ensure that untrusted, foreign-origin Web content included into the hybrid app (for example, ads confined in iframes) cannot access local resources that are available to the app itself.

Our contributions. Our first contribution is to demonstrate that hybrid application frameworks do not correctly compose the same origin policy and the local access-control policy. We will use the term *fracking* for the generic class of vulnerabilities that allow untrusted Web content to drill through the layers of the stack shown in Fig. 1 and reach local resources on the device, thus gaining the ability to steal the user's contacts, write into the device's external storage, manipulate the camera, etc. The technical details differ from framework to framework and from bridge to bridge, but fracking vulnerabilities affect *all* hybrid frameworks on all mobile and desktop platforms, and consequently all hybrid apps based on any of these frameworks.

Our second contribution is a critical analysis of the defenses that hybrid frameworks deploy against fracking attacks. We focus in particular on PhoneGap's origin checks, which are intended to ensure that Web content from untrusted origins cannot invoke certain bridges to local resources. We demonstrate that these defenses are ineffectual because of conceptual mistakes, implementation bugs, and architectural issues in embedded browsers that limit the visibility of the framework's local half into Web code inside the browser. Furthermore, all hybrid frameworks are vulnerable to the *chosen-bridge attack*. Even if the origin check for a particular kind of bridge were implemented correctly, malicious Web content can simply choose another, unprotected bridge and bypass the defense.

We also show that the security properties that hybrid frameworks aim (but fail) to enforce are inconsistent. We call the correct property *NoBridge*: Web content from untrusted origins cannot access the bridge. This is the property that hybrid frameworks promise and that the developers of hybrid apps expect and rely upon. Unfortunately, while promising *NoBridge* to the developers, some frameworks instead enforce a different property we call *NoLoad*: the hybrid app cannot load any Web content from untrusted origins. *NoLoad* is much cruder than *NoBridge* and breaks the business model of many free apps because it prevents them from displaying third-party ads or, in general, any third-party content. Furthermore, even this property is enforced incorrectly.

Our third contribution is a large-scale survey of fracking vulnerabilities, focusing on free, PhoneGap-based Android apps. We chose PhoneGap for our survey because it is currently the most popular hybrid framework and Android because its current market share is over 80%.[1]

Our fourth contribution is a simple capability-based defense called NoFrak. We implemented a prototype of NoFrak as a patch to PhoneGap on Android, but conceptually NoFrak is compatible with any hybrid framework, any embedded browser, and any platform. It enforces the exact security property that hybrid frameworks promise and app developers already rely on. NoFrak requires no changes to the code of the existing hybrid apps and can be deployed transparently, by modifying the framework but keeping the same JavaScript API. Unlike prior defenses, NoFrak is fine-grained and compatible with advertising-supported apps: an app can load third-party content, but this content is blocked from accessing local resources on the device. If necessary, access to specific resources can be granted on a domain-by-domain basis.

II. Hybrid apps and frameworks

Hybrid applications (apps) are a new type of software that is implemented in conventional Web code but also includes a local component intended to execute outside the Web browser on a mobile device or desktop computer (we will refer to the local OS as the *platform*).

Hybrid apps are usually based on one of the hybrid *frameworks* listed in Section II-A. The main reason for the popularity of hybrid frameworks is the support they provide for cross-platform, implement-once-deploy-on-any-OS app development. Hybrid frameworks enable developers to write their apps in portable Web languages such as HTML, JavaScript, and CSS. The framework supplies the app's local component as well as the *bridges* (see Section IV) that allow the app's Web code to escape the browser sandbox and access local resources on the machine, such as the geolocation, camera, contacts, etc., thus relieving developers of the need to write their own device-based code in platform-specific languages such as Java or Objective-C.

The hybrid software stack is opaque and poorly understood by many developers. Hybrid apps delegate security enforcement—in particular, ensuring that foreign-origin Web content included into the app cannot gain access to local resources—to the underlying hybrid framework that creates and manages this stack. Therefore, hybrid apps inherit all of the frameworks' security flaws and vulnerabilities.

A. Hybrid application frameworks

PhoneGap. PhoneGap is a free, open-source framework currently used by over 400,000 developers worldwide. It is the most popular hybrid framework at the time of this writing. PhoneGap supports app development on nine mobile and

[1] http://techcrunch.com/2013/11/12/windows-phone-android-gain-market-share-while-apple-slips-despite-growth-in-iphone-shipments/

desktop platforms, including Android, iOS, Windows Phone, Windows 8, MacOS, and Blackberry. PhoneGap development tools were acquired by Adobe in 2011. Adobe's cloud-based PhoneGap Build enables "develop-once-deploy-everywhere," but if an app is maintained by the developer locally, a separate project is required for each target platform.

By default, PhoneGap-based Android apps request from the user and expose to the app's Web code 16 resources, including camera, fine location, audio and video recording, reading and writing contacts, writing external storage, etc. Individual apps may choose not to request permissions to some or all of these resources. Additional resources can be exposed by third-party PhoneGap plugins.[2]

The latest versions of PhoneGap allow the developer to whitelist authorized Web domains. PhoneGap then tries to ensure that Web content from non-whitelisted domains cannot access the local resources available to the app. In Section VII-B, we explain why this defense is ineffectual.

MoSync. MoSync is a free, open-source framework that provides its own implementation of PhoneGap's JavaScript API. Therefore, hybrid apps designed to work with PhoneGap's API will work with MoSync. MoSync supports "develop-once-deploy-everywhere" without requiring that the app be built in the cloud.

By default, MoSync exposes access to Internet and file storage, but apps can request more permissions if needed. Unlike PhoneGap, MoSync does not appear to provide any way for the app's developer to specify that only certain Web domains may access local resources.

Web Marmalade. Web Marmalade is a proprietary framework currently used by over 50,000 developers on Android and iOS. It provides its own implementation of PhoneGap's JavaScript API and supports "develop-once-deploy-everywhere" via a local IDE, as opposed to cloud build. It does not appear to provide any way for the app's developer to specify that only certain Web domains may access local resources.

appMobi. appMobi is a free framework. In February 2013, appMobi development tools were acquired by Intel. Several parts of the software stack were available as open source in 2011, but as of April 2013, appMobi's git repository is empty. appMobi supports "develop-once-deploy-everywhere" on eight platforms and allows local and cloud build via Intel's XDK.

appMobi provides its own implementation of PhoneGap's JavaScript API and uses a fork of PhoneGap on the local side. An app can specify a domain whitelist via a JavaScript call.

BlackBerry WebWorks. WebWorks is an open-source hybrid app development framework [4] for BlackBerry platforms, including BlackBerry 5.0, BlackBerry PlayBook, and BlackBerry 10. Unlike other frameworks, WebWorks was developed by a platform provider and integrated with a custom embedded Web browser. In contrast, other frameworks employ the platform's default embedded browser, such as WebView on Android, WebBrowser on Windows Phone, etc. (see Section II-B).

WebWorks allows hybrid apps to access local resources such as the camera, microphone, accelerometer, file system, etc. Third-party extensions can expose other local functionalities.[3] Uniquely among hybrid frameworks, WebWorks supports fine-grained, domain-specific access control for local resources (see Section VII-E).

Other hybrid development frameworks include RhoMobile, AppCelerator Titanium, Appspresso, and CocoonJS.

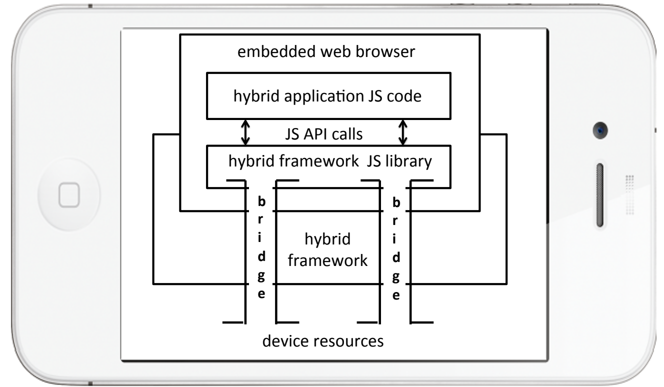

Fig. 2: Components of hybrid frameworks

B. Hybrid software stack

The software stack created by hybrid application frameworks is shown in Figs. 1 and 2. This hybrid software stack is generic and essentially platform-independent, except for minor details (e.g., which embedded Web browser is used on a given OS). The security flaws of this stack, which are analyzed in the rest of this paper, are not specific to a particular browser (e.g., WebView) or a particular way of constructing a bridge. These flaws are generic, too, and affect all platforms, browsers, and frameworks.

The hybrid software stack is a recent entry into the Web and mobile software landscape, and many developers may not fully understand the security implications of combining multiple layers with very different security models (explained in Section III). From the security perspective, the key components of any hybrid framework are the bridges between its Web-facing JavaScript half and its device-based local half, as shown in Fig. 2.

At the bottom of the stack is the OS (e.g., Android, iOS, MacOS, etc.) that manages local device resources such as the camera, file system, location, etc. The local half of the hybrid framework is implemented in Java (Android and BlackBerry), Objective-C (iOS), or C# (Windows Phone). It is incorporated into the hybrid app and runs as a local application from the viewpoint of the OS. When installed on the machine, it obtains access to device resources using one of the mechanisms described in Section III-B.

The framework's local half includes an embedded Web browser. The browser is platform-specific, e.g., WebView in Android, UIWebView in iOS, WebBrowser in Windows Phone,

[2]https://github.com/phonegap/phonegap-plugins/tree/master/Android

[3]https://github.com/blackberry/WebWorks-Community-APIs

or WebWorks in BlackBerry. The framework executes the app's own code within this browser, yet enables this code to access resources as if it were running as a local process on the device. To this end, the framework supplies one or more bridges that bypass the browser sandbox. All bridges include (1) a local component with access to device resources, and (2) a mechanism for the JavaScript code in the browser to invoke the bridge and pass calls and their arguments to the local component. The technical implementation details vary from bridge to bridge (see Section IV).

The hybrid app itself is implemented as Web content in HTML and JavaScript. It executes in the embedded browser but, unlike conventional Web code, can access local resources via bridges added by the framework. Our survey in Section VI shows that hybrid apps often include foreign-origin content, such as ads fetched over HTTP or, rarely, HTTPS.

III. Security models

The primary focus of this paper is on *Web attackers*. A Web attacker controls one or more domains and may host malicious JavaScript there. He does not, however, control the network and cannot tamper with or observe the communications between users and other domains. We assume that the attacker cannot execute his native code on the user's device. Consequently, mobile malware is outside our scope.

Web content served by a hybrid app may include iframes with ads or other content from third parties. These ads are supplied by an ad broker and the app's owner often has no control over their source or content. Therefore, a Web attacker may trick a legitimate app into including malicious content via syndicated advertising. In general, this is not a security hole by itself since foreign-origin content is confined by the browser's same origin policy (see Section III-A).

A stronger threat model is the *network attacker* who can act as a man-in-the-middle on the network between the device and the origins of its Web content. Any content fetched over HTTP is insecure against the network attacker because it can inject arbitrary malicious scripts into any HTTP origin.

A. Web security

The Web content of a hybrid app is governed by the same origin policy (SOP). The origin of Web content is defined by its protocol (HTTP or HTTPS), domain, and port number [2]. SOP is enforced by all embedded Web browsers used in hybrid frameworks (see Section II-B). For example, if an app's Web content includes an iframe with an ad, SOP prevents scripts in this ad from reading or modifying any non-trivial DOM attributes of the app's own content.

Scripts from ad brokers, analytics services, social networks, etc. are usually included directly into the app's Web content and thus run in the app's origin. For example, an app may include a script from an ad syndicator, which creates an iframe and fetches an ad to display within this iframe. The syndicator's script runs in the app's origin, but the actual ad runs in its own origin, isolated from the rest of the app's content by the SOP. Therefore, in our threat model we consider malicious advertisers, but not malicious ad brokers. The latter are trusted by app developers and, critically, their Web code is indistinguishable from the app's own code as far as the SOP is concerned.

B. Local security

The local half of the hybrid app is supplied by the hybrid framework. Its security model is determined by the OS, which mediates access to device resources, and is very different from the Web security model described in Section III-A.

Android uses static permissions (130 as of this writing).[4] An app requests them at the time of installation and the user can either grant all of them, or decline to install the app. Windows Phone 7.1 has 28 static install-time permissions [30]. BlackBerry OS 5 and 6 have 20 permissions in three categories: connections (6), interactions (10), and user data (4) [3]. Unlike other mobile OSes, BlackBerry OS lets the user grant only a subset of permissions requested by an app.

Apple's iOS uses dynamic run-time access control. When an app accesses a resource for the first time, iOS prompts the user to grant the permission. Unlike Android, iOS 6 has only a few permissions. They control access to location, social networks, address book, calendar, Bluetooth, camera, etc.

Fracking vulnerabilities affect all platforms regardless of their access-control model, including desktop platforms like MacOS that do not rely on user-granted permissions. Fracking is more dangerous when permissions are coarse-grained because a single vulnerability exposes more native resources. For example, Windows Phone 7.1 has a single permission for all sensors (ID_CAP_SENSORS). If a hybrid app requests this permission and exposes it to untrusted Web content, the latter will be able to access any sensor on the device.

C. Hybrid security

Security of hybrid apps depends on very subtle trust relationships. For example, on a permission-based platform like Android, the app requests access to device resources from the user. The user grants these permissions to the app—but obviously not to the foreign-origin content (such as ads) included in the app. The app owner trusts the ad broker and includes the broker's scripts into its own origin. These scripts create iframes and display ads in them. Nobody—neither the user, nor the app, nor the ad broker—trusts the advertisers and their content, which is why ads, as well as other foreign-origin content, are isolated in iframes, so that the SOP can block them from accessing other parties' resources.

Hybrid frameworks must guarantee the following security property: **Web content from untrusted domains included into a hybrid app should not be able to access device resources available to the app**. To achieve this, hybrid frameworks must carefully "glue together" two very different security policies: the same origin policy governing the app's Web content, and the OS access-control policy governing the framework's local half. In particular, hybrid frameworks must correctly propagate the same origin policy to local objects *outside* the Web browser when these objects are accessible from inside the browser via the framework's bridges.

[4]http://developer.android.com/reference/android/Manifest.permission.html

In practice, hybrid frameworks attempt to enforce one of the following two properties: Web content from unauthorized domains cannot access the bridges created by the framework (we call this property *NoBridge*), or the app cannot load any content from untrusted domains (we call this property *NoLoad*). Both properties prevent undesired access to resources, but *NoLoad* breaks the business model of many free apps because it prevents them from displaying foreign-origin ads. In Section VII, we show that the frameworks' defenses are inconsistent and apply different policies in different parts of the same code, often incorrectly.

IV. Bridges

The main purpose of any hybrid framework is to supply *bridges* that enable the app's Web content to access local device resources. The app, however, does not invoke these bridges directly. Instead, it uses the framework's JavaScript library, which presents a structured, platform-independent API for resource access. This library invokes the bridges internally, thus the mechanics of different bridge architectures are transparent to the app and the app's code does not directly depend on the specific bridges used by the framework.

To enable asynchronous resource access, frameworks often provide separate Web-to-local bridges (for invoking a local function) and local-to-Web bridges (for receiving the result). With this separation, JavaScript executing in the embedded browser does not block while the device is accessing the resource.

A. Web-to-local bridges

Interfaces. Several embedded browsers allow local code to expose arbitrary functions to JavaScript running within the browser. For example, 'addJavascriptInterface' in Android's WebView makes local Java objects visible to JavaScript. Other platforms have similar functionalities, for example, 'windowScriptObject' in MacOS and 'ScriptEngine.addExtension' in BlackBerry WebWorks.

This technique is used, among others, by PhoneGap, AppCelerator Titanium, and Sencha Touch to enable direct access to local device resources by Web content.

On Android prior to API level 17, these interfaces are generically insecure. Malicious JavaScript executing inside WebView can use the Java reflection API to invoke any method of any Java object exposed via 'addJavascriptInterface' and take control over the local side of the application [1, 18, 23]. Starting from Android API level 17, only the methods explicitly annotated with @JavascriptInterface are visible in the Java objects exposed to WebView via 'addJavascriptInterface'.

Events. Another common mechanism for the local half of the framework to receive messages from the Web half is via various JavaScript events. To use this mechanism, the local half must override the event handlers in the embedded browser. JavaScript on the Web side triggers events and encodes messages in arbitrary strings, the handlers intercept these events and decode the messages.

For example, WebView, Android's embedded browser class, allows local Java code to customize the handling of 'prompt', 'alert', and 'confirm' events by overriding the 'onJsPrompt', 'onJsAlert', and 'onJsConfirm' functions, respectively. This bridge mechanism is used by PhoneGap on Android 2.3 because of a bug[5] that precludes the use of an interface-based bridge described above. On the local side, PhoneGap implements a custom 'onJsPrompt' handler. On the Web side, JavaScript makes a 'prompt' call to invoke this handler; the name and the arguments of the local object to be called are serialized and sent to the local side by encoding them as prompt messages. MoSync on Android uses a similar mechanism.

Similarly, WebBrowser, the embedded browser class in Windows Phone, allows local C# code to install custom handlers for 'ScriptNotify' events. JavaScript on the Web side uses 'window.external.Notify' to trigger these events and invoke local functions. This technique is used by PhoneGap on Windows Phone.

Event-based bridges can be synchronous (JavaScript pauses until the call returns) or asynchronous (the call returns a placeholder or null, but the actual data is returned later via a local-to-Web bridge).

URL loading interposition. Embedded browsers typically allow local code to interpose on and handle URL loading within the browser. This feature is intended to support implementation of custom protocols, but hybrid frameworks overload it so that the Web half of the framework can pass arbitrary messages to the local half by encoding them in the URL, thereby constructing a Web-to-local bridge.

URLs intended for interposition cannot be loaded in the main browser frame lest the entire app blocks. Instead, hybrid frameworks use one of the following two methods.

JavaScript on the Web side can create an invisible iframe and set its source to an arbitrary URL. The loading of this URL is intercepted by a handler supplied by the framework's local half, without affecting the execution of the main frame on the Web side. The handler decodes the URL and passes the message to the local half. For example, PhoneGap on iOS creates its own subclass of NSURLProtocol named CDVURLProtocol and calls registerClass function to intercept URLs loaded in the UIWebView embedded browser. PhoneGap's JavaScript library creates an iframe whose URL starts with gap://, PhoneGap's local half intercepts the loading of this URL and calls the corresponding local function. In Web Marmalade,[6] the framework's JavaScript library creates an iframe with the src attribute s3ebridge://queued. A shared JavaScript object, s3e.queue, in this iframe is used for communication between the Web half and local half.

As an alternative to iframe-based communication, JavaScript on the Web side can make an asynchronous XMLHttpRequest to a URL which is intercepted and decoded by the framework's local handler. PhoneGap uses this technique on iOS in addition to the iframe URL interception.

[5] http://code.google.com/p/android/issues/detail?id=12987

[6] https://github.com/marmalade/Web-Marmalade/blob/master/wmClipboard/data/webassets/wm.js

Cookies (obsolete). The framework's Web half can set cookies that are readable by the local half, and vice versa. PhoneGap used this technique on older platforms like BlackBerry 4.x,

B. Local-to-Web bridges

Local-to-Web bridges are used by the framework's local half to return data to the framework's JavaScript library on the Web side. Some data such as device ID can be returned synchronously via the function's return value, but usually the results of device access (e.g., contact list, local files, etc.) are returned asynchronously, to avoid blocking Web code while access is performed.

Multiplexing a Web-to-local bridge. Messages from the local half to the Web half can be sent over the same bridge as the Web-to-local messages. There are two asynchronous mechanisms for doing this.

Local code can trigger events to notify the JavaScript library that there is data waiting for it. JavaScript then reads the data via one of the Web-to-local bridges from Section IV-A. This mechanism, using online/offline events,[7] is employed by PhoneGap on Android.

On many platforms, event-based synchronization is not available if the framework's JavaScript is running inside an iframe. Many embedded browsers—including WebView on Android, UIWebView on iOS, and WebBrowser on Windows Phone—incorrectly deliver some events to the main frame even if the handler is registered inside an iframe. This issue is sometimes referred to as "frame confusion" [17].

The alternative is to have JavaScript poll the Web-to-local bridge at periodic intervals to check whether data is ready. PhoneGap on Android supports this technique, too.

Interfaces. Many embedded browsers have helper functions that let local code execute arbitrary JavaScript inside the browser. For example, Android's WebView has a private reflection API, as well as the 'loadUrl' function that allows Java code to load a javascript: URL inside the browser. PhoneGap on Android uses both. PhoneGap on iOS, BlackBerry, and Windows Phone uses 'stringByEvaluatingJavaScriptFromString', 'ScriptEngine.executeScript', and 'WebBrowser.InvokeScript' functions, respectively, to execute JavaScript inside the corresponding embedded browsers.

C. Custom cross-origin communication

As mentioned above, many local-to-Web bridges often do not work if the framework's JavaScript library is running inside an iframe. For example, on iOS and Windows Phone 7, bugs in embedded browsers cause the data to be returned to the main frame rather than the iframe that invoked the bridge.

This forces app developers to implement "home-brewed" cross-origin mechanisms to deliver the data from the device to the iframe that needs it.[8] [9] [10] These custom, hand-coded hacks open holes in the same origin policy and can introduce serious security vulnerabilities, including cross-site scripting (e.g., see [25]). While interesting, this class of vulnerabilities is not specific to hybrid apps and we leave its detailed analysis to future work.

V. FRACKING

As explained in Section III-C, the hybrid framework must guarantee that untrusted foreign-origin content included into the app—for example, ads—not be able to access local device resources such as the file system, contacts, SMS, etc. via the bridges provided by the framework. Unauthorized access can be exploited for privacy violations (e.g., steal the user's contacts list), security breaches (e.g., use personal data for targeted phishing), monetary gains (e.g., send SMS to premium numbers), or denial of service (e.g., cause the phone to vibrate or beep incessantly).

To prevent unauthorized resource access by foreign-origin Web content, hybrid frameworks rely on several layers of defense. The first layer is the same origin policy (SOP) within the embedded browser (see Section III-A): for example, ads are confined within iframes and thus prevented by the SOP from accessing bridges in other frames. The second layer is the bridge mechanism itself which may be difficult to invoke from inside an iframe. The third layer is the origin checks used by the framework's local half to block bridge access initiated by unauthorized Web content (see Section VII-B). The fourth layer is access control within the operating system (see Section III-B).

A. Conceptual overview

We use the term *fracking* for any attack that allows malicious foreign-origin JavaScript to "drill" through the defense layers and gain unauthorized access to device resources. Several features distinguish fracking vulnerabilities from other security holes in hybrid and native mobile apps.

First, fracking attacks exploit the mismatches and semantic gaps between the SOP-based security policy governing the Web half of hybrid apps and the OS access-control policy governing their local half—in particular, the hybrid frameworks' failure to correctly apply the SOP to bridges. Consequently, fracking vulnerabilities are generic and affect all bridge mechanisms in all hybrid frameworks, all embedded browsers, and all mobile and desktop platforms.

Second, fracking vulnerabilities are not caused by the well-known weaknesses of access control based on user-granted permissions, such as the tendency of apps to request too many permissions or the failure of users to understand the consequences of indiscriminately granting permission requests. Fracking affects hybrid frameworks even on desktop platforms such as MacOS where access control is not based on user-granted permissions. Even on Android, the problem is not simply that the app requests too many permissions (in fact,

[7]These events are sent by HTML5 browsers to Web pages when the browser detects the Internet connection to be enabled/disabled; PhoneGap re-purposes them for local-to-Web notifications.

[8]http://comments.gmane.org/gmane.comp.handhelds.phonegap/16406

[9]http://stackoverflow.com/questions/5875083/possible-to-access-phonegap-api-within-a-iframe

[10]http://hackerluddite.wordpress.com/2012/04/15/getting-access-to-a-phones-camera-from-a-web-page/

many legitimate hybrid apps do need these permissions in order to function correctly), but that these permissions get exposed to untrusted Web content included in the app.

Third, fracking is not an instance of Android's permission re-delegation problem [7, 9]. Fracking vulnerabilities occur at a different layer of the software stack, where permission-based local access control meets origin-based Web access control. Furthermore, hybrid frameworks such as PhoneGap do not blindly delegate their permissions to untrusted Web content. They deploy several defenses to prevent "confused deputy" [14] attacks and to ensure that local permissions are only used from the origin to which they were delegated. Unfortunately, in Section VII we show that these defenses are largely ineffectual.

Fourth, there is an important distinction between conventional and embedded Web browsers that makes defending against fracking difficult. Conventional browsers interact with users. For example, Chrome's implementation of the WebRTC API [29], which allows Web content to access local resources such as camera and microphone, pops a user dialog box every time an access is attempted. This dialog shows the name of the requesting domain and asks the user for permission. In contrast, hybrid frameworks use embedded browsers so that hybrid apps can *automatically* access local device resources. They cannot afford to ask the user for permission on every access and must rely on programmatic defenses.

B. Technical details

A fracking attack is performed by a malicious script confined within an iframe in the embedded browser. SOP prevents it from accessing objects in other frames, but does not stop it from including the hybrid framework's JavaScript library or even its own, arbitrarily modified clone of this library, crafted to maliciously access the framework's bridges.

Chosen-bridge attacks are devastating in this setting. Frameworks like PhoneGap support multiple bridges to and from local resources. Furthermore, they allow JavaScript on the Web side to choose a bridge via 'setNativeToJsBridgeMode' and 'setJsToNativeBridgeMode' functions. These functions are not intended to be called directly by hybrid apps, since apps are supposed to access the framework only through the public API, but they are not protected by the SOP. Therefore, a malicious script is free to invoke them in order to select a vulnerable bridge. Consequently, even if some bridges are secure, a single vulnerable bridge is sufficient to bypass all of the framework's defenses. Availability of bridges varies from version to version even within the same framework, but malicious JavaScript can determine the version (e.g., via 'device.cordova' in PhoneGap and MoSync) and choose the attack adaptively.

Exploiting interface-based bridges. Any JavaScript object added to the embedded browser by the framework's local half via functions such as 'addJavascriptInterface' in Android's WebView or 'ScriptEngine.addExtension' in BlackBerry is available by default to JavaScript in any iframe, regardless of its origin.

Frame confusion complicates the exploitation of interface-based local-to-Web bridges on some platforms. The 'stringByEvaluatingJavaScriptFromString' and 'WebBrowser.InvokeScript' functions, used by the framework's local half on iOS and Windows Phone, respectively, to inject JavaScript into browsers, execute it in the main frame, not the iframe that invoked the bridge. Therefore, malicious JavaScript inside an iframe cannot see data returned from the device, but can still cause malicious side effects through Web-to-local bridges without seeing the return values, e.g., create or delete contacts, send SMS to premium numbers, etc.

Exploiting event-based bridges. Event-based local-to-Web bridges are difficult to use if the framework's JavaScript library is running inside an iframe. The events that synchronize the framework's Web and local halves are always delivered to the main frame, even if the handler had been registered from an iframe, thus preventing the script in the iframe from learning that the local half is ready with the results (see Section IV-B). Furthermore, some of the utility JavaScript objects created by the framework are not accessible to JavaScript inside iframes. Because of this, some analyses mistakenly concluded that event-based bridges cannot be exploited by malicious JavaScript [19].

This conclusion is false. Modified, malicious clones of the framework's JavaScript library can access local resources via event-based bridges even when confined in an iframe.

First, if the malicious script inside an iframe cannot receive synchronization events from the framework's local half, it can simply block for a predefined interval until the local half is ready. In event-based bridges on Android, the Java side transfers data to JavaScript through the return values of the 'OnJsPrompt' handler. Unlike synchronization events, these values are correctly delivered by the embedded browser to malicious JavaScript code inside the iframe.

Second, even if the framework's utility objects are not visible from an iframe, the main JavaScript objects implementing the bridge are available, and malicious code can access them directly. For instance, if malicious JavaScript wants to access the contact list on an Android device via a PhoneGap bridge, it can (1) call cordova.require('cordova/exec') to obtain a reference to the exec function that invokes the bridge, and (2) call cordova.require('cordova/plugin/ContactFindOptions') to obtain a reference to the contacts search filter. The rest of the code can be cloned from the framework's own JavaScript library and will work without modifications.

Exploiting URL interposition-based bridges. Both methods for asynchronous URL loading—fetching an invisible iframe whose source URL encodes the message or issuing an XMLHttpRequest to an encoded URL—work from inside any iframe. Malicious JavaScript confined in an iframe can use either bridge to access the framework's local half.

VI. Fracking in the wild

To estimate the prevalence of fracking vulnerabilities in real-world hybrid applications, we analyzed 7,167 free Android apps built using PhoneGap, currently the most popular hybrid framework. These apps were identified in our dataset of 128,000 free apps collected from Google Play between January 18 and March 18, 2013, by the presence of "cordovaVersion" or "phonegapVersion" in the dexdump of their APK (file

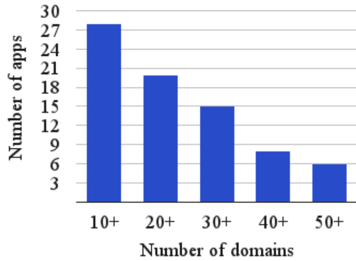

Fig. 3: Read Contacts

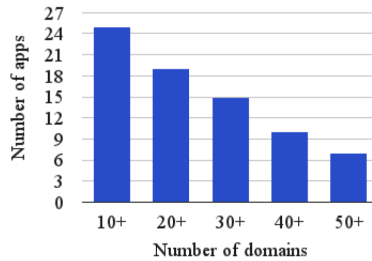

Fig. 4: Write Contacts

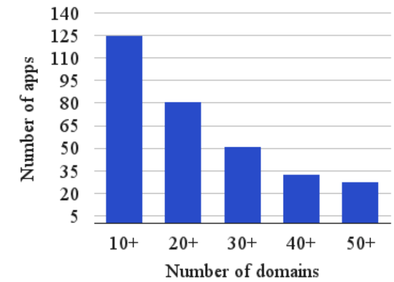

Fig. 5: Write External Storage

format for Android apps) and the presence of PhoneGap plugins in "plugins.xml", "cordova.xml", or "phonegap.xml".

We implemented a tool called GapFinder to automatically extract from each PhoneGap app its (1) local permissions, (2) a subset of the Web domains whose content is included in the app, and (3) the domain whitelist, if any. To extract permissions, GapFinder uses Android's aapt (Android Asset Packaging Tool). To find domains contributing Web content to the app, GapFinder uses apktool to reverse-engineer the APK, extracts HTML files used by the app, and automatically crawls each HTML file to depth 1 using Selenium with a Google Chrome driver and the browser string overwritten to mimic a Nexus 4 device. The resulting set of domains is a conservative underestimate because the app may fetch content from additional domains reached only by browsing to depth 2 or deeper. Also, with syndicated advertising, the actual domains may change from execution to execution.

3,794 apps do not include any external content (such as iframes, scripts, etc.) in the HTML files extracted by our tool. 45 apps include only HTTPS content. The remaining 3,328 apps include content from at least one external domain via HTTP. Their device resources are thus potentially vulnerable to both Web attackers (hosted at any of these domains) and network attackers.

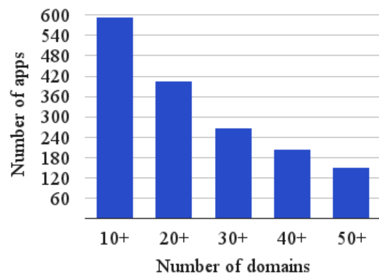

Fig. 6: Access Fine Location

Figs. 3 through 6 estimate the exposure of specific sensitive resources to Web attackers. For example, Fig. 3 shows that 20 PhoneGap apps expose the ability to read the contacts list on the device to 20 or more Web domains each. Fig. 4 shows that 19 apps expose the ability to write the contacts list to 20 or more domains each. Fig. 5 shows that each of 81 apps allows at least 20 domains to write into the device's external storage. Fig. 6 shows that 407 apps expose fine-grained location data to at least 20 domains each.

All vulnerabilities were empirically confirmed by randomly sampling the set of vulnerable apps, injecting attack JavaScript into one or more foreign origins included into the app, and verifying that this JavaScript can successfully access local device resources.

Figs. 7 and 8 estimate the extent of exposure, i.e., how many Web domains can access device resources by fracking a hybrid app. Fig. 7 shows that more than 150 apps expose their resources to at least 50 Web domains each. Fig. 8 shows that most external domains have access to between 4 and 6 resources, but some have access to all 16 resources available in default PhoneGap.

Fig. 9 demonstrates that many apps use third-party plugins. These plugins expose many more resources than default PhoneGap, including fine-grained social-media permissions such as access to the user's Facebook and Twitter. More than half of the apps in our survey incorporate at least 10 different plugins.

Examples of vulnerable PhoneGap apps include *ForzeArmate*, an app for Italian servicemen which exposes the ability to write into external storage (among other permissions) to any domain advertising through Google syndication; the *Edinburgh by Bus* app, which exposes external storage to a large number of obscure Eastern European dating sites; and *DrinkedIn BarFinder*, which exposes fine geolocation to domains such as freelifetimecheating.com, www.babesroulette.com, and many adult sites. Furthermore, content from all of these domains is loaded over HTTP and thus vulnerable to network attackers, who automatically gain access to the app's local permissions.

VII. Existing defenses

A. *Relying on the browser*

Several hybrid frameworks, including MoSync and Web Marmalade, rely on the embedded browser to prevent untrusted Web content from accessing bridges to local resources. Because the bridges are added to the browser by local code, they have no Web origin as far as the browser is concerned. Therefore, malicious Web content from any origin can directly invoke the bridges, as explained in Section V-B.

All hybrid apps based on any of these frameworks are generically vulnerable to fracking.

B. *Whitelisting authorized domains*

Some hybrid frameworks, including PhoneGap and BlackBerry WebWorks, implement defenses specifically to prevent foreign-origin content from accessing bridges (i.e., fracking attacks). The app creator supplies a *whitelist* of authorized domains, e.g., the app's own domain. In PhoneGap, the whitelist

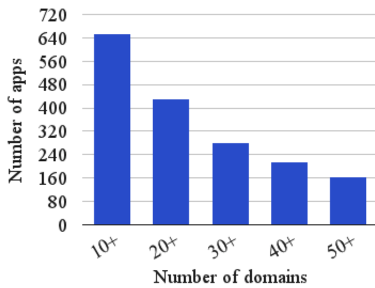

Fig. 7: Foreign-origin content

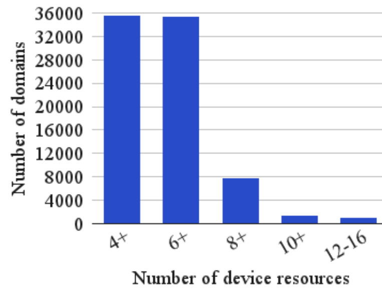

Fig. 8: Exposure of device resources to foreign origins

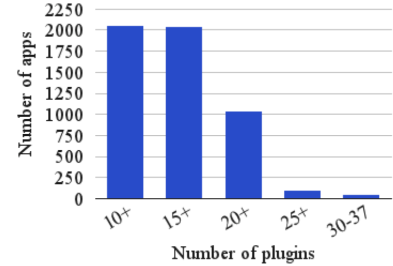

Fig. 9: Hybrid apps with plugins

is specified via a platform-specific configuration file such as 'cordova.xml' or 'config.xml'.

Correctly implementing this defense is subtle and error-prone. The right property is *NoBridge* (Section III-C): Web content loaded by the hybrid app from a non-whitelisted origin should not be able to access the bridge. Instead, the properties enforced by the actual defenses differ from framework to framework, platform to platform, and even from bridge to bridge within the same framework. For example, some PhoneGap bridges aim to enforce *NoBridge*, while other parts of the same framework aim to enforce a property we call *NoLoad*: the hybrid app should not be able to load any Web content from a non-whitelisted origin. Obviously, *NoLoad* implies *NoBridge*, but it is much coarser and breaks both the user interface and the business model of many free apps.

C. Enforcing NoLoad

PhoneGap on Android, iOS, and Windows Phone attempts to enforce the *NoLoad* property: if the app fetches an HTML element (e.g., an iframe) from a non-whitelisted domain, the element is simply not loaded. Main frames from non-whitelisted domains are opened in the default system browser which does not have any bridges. Interestingly, event-based bridges in PhoneGap on Android attempt to enforce the *NoBridge* property (see Section VII-E), too, even though *NoLoad* implies *NoBridge* and, had it been implemented correctly, would have rendered the *NoBridge* checks in the same code superfluous.

Implementations of the *NoLoad* defense are rife with errors of two major types: **incorrect URL interception** and **incorrect URL matching** against the domain whitelist.

Android (other than event-based bridges). Before version 2.6, PhoneGap on Android used a WebView callback 'shouldOverrideUrlLoading' to intercept the loading of foreign-origin content. This callback is not invoked for iframe fetches or XMLHttpRequests. Therefore, this defense cannot prevent a hybrid app from loading unauthorized content as, for example, an ad in an iframe. PhoneGap 2.6, released on April 9, 2013, uses the 'shouldInterceptRequest' callback which correctly intercepts the loading of iframes. This callback is only supported by Android API 11 or later.

Unfortunately, this implementation intercepts and blocks the loading of *any* content from non-whitelisted domains. The problem is that URL interception in WebView does not provide any way to distinguish between URLs loaded in the same origin (e.g., via ⟨script⟩ tags) and URLs loaded in a foreign origin (e.g., via ⟨iframe⟩ tags). The resulting policy is thus *stricter* than the standard same origin policy! The hybrid app is not only prevented from loading foreign-origin content, but it cannot even include an external image or JavaScript library, which is a very common practice.

All tested versions of PhoneGap for Android, including 2.6, *incorrectly match intercepted URLs against the whitelist*. PhoneGap uses Java's regular expression engine and anchors the expression for each whitelisted domain only at the beginning, but not the end:

```
this.whiteList.add(Pattern.compile("^https?://(.*\\.)?" +
     origin));
```

For example, if foo.com is whitelisted, PhoneGap allows content to be loaded from foo.com.evil.com, violating the desired property. A similar observation was made in [19].

HTTP/HTTPS is ignored when checking URLs against the white list. A network attacker can thus downgrade connections from HTTPS to HTTP and inject malicious scripts into whitelisted origins.

iOS. PhoneGap creates a subclass of NSURLProtocol named CDVURLProtocol to intercept URLs loaded in UIWebView and check whether the URL is whitelisted. UIWebView suffers from the same problem as WebView and the resulting policy is stricter than the same origin policy.

PhoneGap on iOS only allows domain names to be specified in the whitelist file, but not HTTP/HTTPS schemes. This prevents the app creator from specifying that certain domains should be loaded only over HTTPS, which is a very important property (see Section VIII-A), and opens the door to HTTPS downgrade attacks.

Windows Phone 7 and 8. PhoneGap installs a handler for the browser's navigation event and checks the whitelist before allowing navigation. This event is not triggered for iframe fetches and XMLHttpRequests. Therefore, this defense fails to guarantee *NoLoad*.

D. Inadequacy of NoLoad

In addition to the fact that virtually all implementations of *NoLoad* are incorrect, the property itself is too coarse. It does not allow the hybrid app to include content from third parties whose domains are not known at the time of compilation. Of course, the entire business model of free, ad-supported hybrid apps relies on their ability to load content, such as

Web advertising, from origins that are determined at runtime. The ad broker's origin may be known statically, but only the script creating the iframe comes from that origin. The actual content inside the iframe comes from an advertiser and its origin is often determined dynamically, e.g., via a real-time auction conducted by the ad broker. **Even if implemented correctly, the *NoLoad* defense is incompatible with the business model of most free apps.**

Suppressing foreign-origin content may also have a negative effect on the app's look-and-feel, creating unsightly white spots where ads and other foreign-origin content would have been loaded (see Fig. 10).

In practice, this defense requires the app creator to make a binary decision whether to whitelist *all* foreign origins—and thus expose the app to fracking attacks—or not use foreign-origin content at all and thus give up on any revenues from Web advertising and, in the case of PhoneGap on iOS and Android (after version 2.6), even analytics and revenue optimization services. Not surprisingly, out of 7,167 hybrid PhoneGap apps in our study, 2,124 whitelist all domains and would have been vulnerable to fracking even if PhoneGap's implementation of whitelisting had been correct.

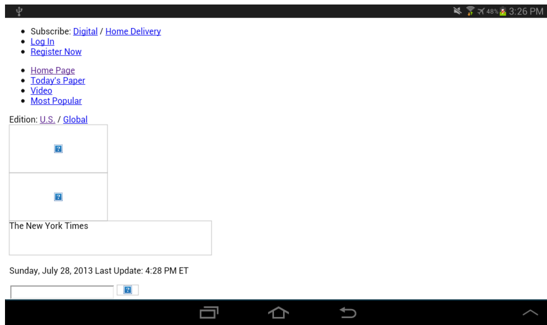

Fig. 10: NY Times with only nytimes.com whitelisted (NoLoad policy)

Fig. 11: NY Times with only nytimes.com whitelisted (NoBridge policy)

E. Enforcing NoBridge

Unlike *NoLoad*, *NoBridge* allows the app to load foreign-origin content, but ensures that only the content from whitelisted domains can access the bridge. *NoBridge* is compatible with the advertising-supported business model of free apps and allows them to preserve their look-and-feel (Fig. 11).

Implementing *NoBridge* critically depends on the ability of the framework's local half to correctly determine the origin of the content that invoked the bridge, in order to match it against the whitelist. Unfortunately, many embedded browsers do not support this. For example, if an interface is added to WebView via 'addJavascriptInterface', the local half cannot determine the origin of the script that called this interface [17].

NoFrak, our capability-based defense described in Section VIII, enforces *NoBridge* without relying on the embedded browser to transmit correct origin information to the framework's local half.

Android (event-based bridges). For event-based bridges only, PhoneGap on Android attempts to enforce the *NoBridge* property. This is possible because, unlike interface-based bridges, event-based bridges preserve the origin of the request. For example, when the bridge is called via the prompt() method (Section IV-A), PhoneGap applies Config.isUrlWhiteListed() to the origin of the script that triggered the prompt.

Unfortunately, the actual check is incorrect because of the anchoring bug described in Section VII-C. If foo.com is whitelisted, malicious JavaScript hosted at any domain starting with foo.com, such as foo.com.evil.com, is permitted to access the bridge.

BlackBerry WebWorks. BlackBerry WebWorks incorporates a custom, WebKit-based embedded browser, which correctly intercepts URLs of HTML elements and XMLHttpRequests. WebWorks is unique in that it can restrict access to specific resources on a domain-by-domain basis.[11] For example, an app creator can use the following whitelist to allow any domain to be loaded within the browser, but only permit '*mydomain.com*' to access the user's contact information:

```
<access uri="https://mydomain.com" subdomains="true">
<feature id="blackberry.find"          ... />
<feature id="blackberry.identity"      ... />
<feature id="blackberry.pim.Address"   ... />
<feature id="blackberry.pim.Contact"   ... />
</access>
<access uri ="*"/>
```

Unfortunately, PhoneGap on BlackBerry does not take advantage of this facility and enforces *NoLoad* rather than *NoBridge* [22].

Because the enforcement of *NoBridge* in WebWorks relies on a customized browser, it is not portable. In contrast, our defense, NoFrak, is platform-independent and compatible with any common embedded browser, including WebView and UIWebView.

F. Relying on local privilege separation

Privilege separation has been proposed in the research literature as a solution to the problem of protecting device resources from untrusted Web content (see Section IX). Privilege separation is straightforward to apply to "pure" mobile apps that incorporate ads via local libraries such as AdMob. The library and its ads are confined into a separate browser instance, while resources are exposed only to a different instance containing the app's own code [26].

[11] https://developer.blackberry.com/html5/documentation/Access_element_834677_11.html

Local process separation is non-trivial for hybrid apps. Unlike mobile apps that display their content and foreign-origin ads in separate browser instances, hybrid apps "mash up" content from multiple origins and render it within a *single* browser instance created by the hybrid framework. Because the entire app acts as a single local process on the device, there is no natural way to extract foreign-origin content and display it in a separate browser.

PhoneGap's loading of non-whitelisted domains in the default system browser is a form of privilege separation since the default browser does not have the bridges added by PhoneGap. It is applied only to main frames, however. Applying this defense to HTML elements like iframes would have required the framework to correctly compose multiple browser instances in order to keep the app's user interface intact. To the best of our knowledge, this complex functionality is not supported by any hybrid framework.

Applying privilege separation along the lines of [24, 26] requires either re-factoring the entire application, or significant modifications to the existing browsers so that they spawn a separate browser instance for each occurrence of foreign-origin content. The latter is not feasible. The former is incompatible with the *raison d'être* of hybrid development frameworks. They are popular precisely because they allow developers to easily convert Web apps into mobile apps and to add native access to Web apps with *minimal modifications* to the app's code and without requiring the creator to completely re-factor her app.

VIII. NoFrak

A. Design

NoFrak is a generic defense against fracking attacks. Its main design principle is to extend origin-based access control to local resources outside the Web browser. To achieve this, NoFrak enforces the *NoBridge* property (Section VII-E): a hybrid app can freely include Web content from any origin, but unauthorized origins cannot access device resources available to the app. This is the exact security property that hybrid frameworks promise to app developers and that all existing hybrid apps already expect and rely upon. Furthermore, it is compatible with the business model of advertising-supported hybrid apps. It permits them to show foreign-origin ads, yet ensures that these ads cannot get unauthorized access to the device.

The key idea behind NoFrak is that all accesses to bridges from the Web side must be authenticated by unforgeable capability tokens. Each token is unique to a particular Web origin and kept in this origin's localStorage. NoFrak thus leverages the browser's same origin policy to ensure that content from other origins cannot read the token and thus cannot access the bridge.

NoFrak does not change the JavaScript API that hybrid apps use for resource access. Therefore, it is fully transparent to all existing hybrid apps, requiring no changes to their code.

On the Web side of the hybrid framework, NoFrak makes a minor modification to the framework's JavaScript library to (1) read the capability token from localStorage, and (2) pass it as an extra argument to every bridge call. This modification is invisible to the apps. Because the extra argument is added before the arguments are marshalled for the bridge call, the implementation of NoFrak does not depend on the specifics of the bridge architecture, which resources are accessed via the bridge, etc. On the local side of the hybrid framework, NoFrak makes a minor modification to check the capability token before permitting resource access.

The design of NoFrak is compatible with every existing platform, hybrid framework, and bridge architecture. As mentioned above, NoFrak does not require any changes to apps' code, but apps must be recompiled because NoFrak changes the local half of the framework, which serves as the local side of each hybrid app.

Whitelist policies. For transparent compatibility with the existing hybrid frameworks such as PhoneGap, NoFrak uses the same interface for specifying which origins are authorized to access local resources: a domain whitelist provided by the app developer.

In PhoneGap, these whitelists are coarse-grained. For example, there is no way for a PhoneGap app to say that content from a particular domain is allowed to access geolocation only. This is a limitation of all hybrid frameworks except BlackBerry WebWorks (see Section VII-E) and has nothing to do with NoFrak *per se*. If needed, NoFrak can also support finer-grained policies and grant access to specific resources on a domain-by-domain basis, as opposed to the blanket authorization for all whitelisted domains to access any resource available to the app. For example, all domains can be authorized to access geolocation, but only the app's own domain is authorized to access the camera. This requires changes to PhoneGap's whitelisting language. Since thousands of apps already rely on the current language, this is not a backward-compatible modification.

Unlike PhoneGap, NoFrak by default does not allow "*" whitelists, but, in contrast to PhoneGap, this does not prevent the app from displaying content from any origin. Most hybrid apps in our survey access device resources only from their local HTML files (those shipped with the app), not from dynamically loaded Web content. This policy change is transparent to them.

Some app developers may wish to express policies like "ads are allowed to access geolocation, but not any other local resource" without explicitly enumerating all domains that may host advertising. Such policies cannot be enforced by any existing Web browser. All access-control decisions for Web content are based on its origin, defined by the protocol, domain, and port number (Section III-A). Because the purpose of NoFrak is to extend origin-based access control to local resources, any policy enforced by NoFrak must be based on the origin of the Web content that attempts to access the resource. Without a fundamental re-engineering of the entire Web security model, it is not possible to restrict the access rights of Web content based on what it does (e.g., advertising) vs. where it comes from (i.e., its origin).

Preventing network attacks. The same origin policy cannot protect a hybrid app from network attacks. If any content from a whitelisted origin is retrieved over HTTP, a man-in-the-middle attacker—for example, a malicious Wi-Fi access

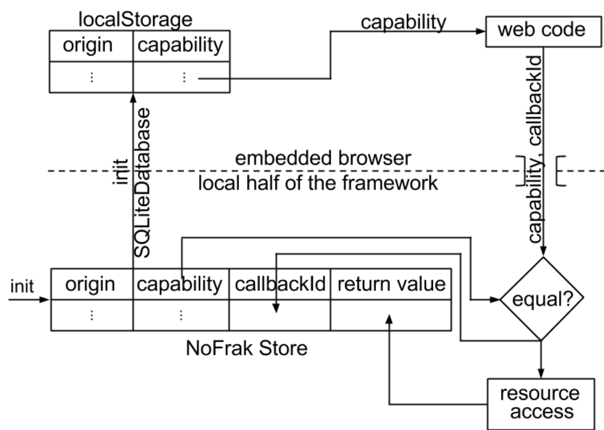

Fig. 12: NoFrak: Invoking a bridge

point—can inject an attack script into it. This script will be treated by the browser as if it came from the whitelisted origin and can thus read this origin's localStorage and obtain the capability token.

To prevent network attacks, the app must whitelist only HTTPS origins. NoFrak then inserts the corresponding tokens into localStorage, and the browser enforces that these tokens can be read only by scripts fetched over HTTPS from the correct origin.

B. Implementation

As a proof of concept, we added NoFrak to PhoneGap on Android. We chose PhoneGap because it is open source and by far the most popular hybrid framework. Our prototype implementation is available for download at https://github.com/georgiev-martin/NoFrak

Our prototype adds 393 lines of code to PhoneGap's Java code and modifies 6 lines of code in PhoneGap's JavaScript library. Any other hybrid framework can be modified in the same way.

Initialization. Like stock PhoneGap, NoFrak accepts a whitelist of authorized domains from the app's developer. The capability for each whitelisted domain is a 9-digit pseudorandom token, generated by calling SecureRandom when the app's local half is initialized. It is kept in the NoFrak Store on the local side. Before the embedded browser instance is created, each token is injected into the browser's localStorage for the corresponding domain.

Web-to-local bridges. The architecture of NoFrak Web-to-local bridges is shown in Fig. 12. Just like with stock PhoneGap, the app's Web code has to include the NoFrak JavaScript library. The API of this library is the same as PhoneGap, thus the app's own code need not be modified.

Internally, the library uses the capability token when accessing any of the available bridges. First, it reads the token via `window.localStorage.getItem(''SecureToken'')`. Scripts from other origins cannot read it because of the same origin policy. To access any bridge, the library calls `exec(service, action, callbackId, args, localStorage.getItem("SecureToken"));`. The local half of NoFrak receives the call, matches the token against the NoFrak Store, and, if found, executes the request

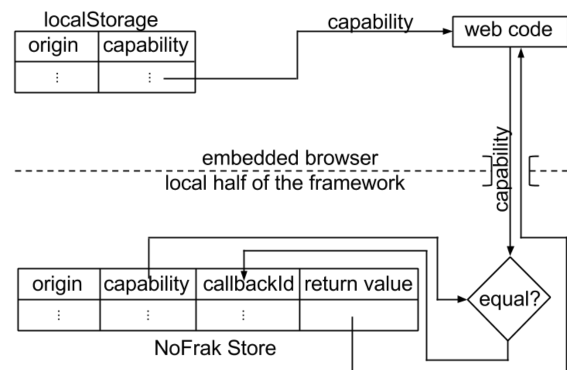

Fig. 13: NoFrak: Retrieving the result

(e.g., accesses a device resource). The NoFrak Store does not have a JavaScript interface and cannot be written to from the Web side, thus NoFrak is immune to localStorage poisoning.

Preventing reflection attacks. As mentioned in Section IV-A, prior to Android API level 17 all bridges based on 'addJavascriptInterface' were vulnerable to reflection attackss [1, 18, 23]. Our prototype implementation of NoFrak is built as a patch to PhoneGap 2.9, which is designed for Android API level 17 and thus immune to reflection attacks.

To enable NoFrak-based apps to run on earlier versions of Android, NoFrak makes two small changes to the local PhoneGap code. First, it sets the default bridge mechanism to events rather than interfaces. Second, it modifies the existing code in PhoneGap's exposeJsInterface() so that it does not add JavaScript interfaces to WebView if the API level is less than 17 (PhoneGap's current code does not add JavaScript interfaces if the API level is less than 9 or equal to 11). This change is backward-compatible and transparent to all benign apps because they only access bridges through the PhoneGap JavaScript API, which remains unchanged. The framework's JavaScript library simply "re-routes" the calls to a different, secure bridge.

To prevent malicious JavaScript from crashing the local side of the app by switching to a non-existing interface bridge, NoFrak also modifies the setNativeToJsBridgeMode() method in the local half to deny the request if the API level is less than 17.

Local-to-Web bridge. The local-to-Web bridge can be synchronous or asynchronous. Synchronous bridges are used for local accesses that can be answered immediately, for example, device or network info. These can reuse the already authenticated Web-to-local bridge, with the response passed to the Web side simply as the return value of the call. Local accesses that require more work, such as camera and media, need to be asynchronous to avoid freezing the app's Web content while the device is being accessed.

Because of the bugs in embedded browsers, events and scripts injected by the local code into the browser can execute in the wrong origin (see Section IV-B). Instead, both asynchronous local-to-Web mechanisms supported by NoFrak reuse the Web-to-local bridge for retrieving the results of bridge invocation.

The first mechanism is a modification to *pending message notification* used by PhoneGap. When the response is ready, the local half of NOFRAK sends an unauthenticated notification to JavaScript on the Web side. The message does not contain any data and is safe even if the browser mistakenly delivers it to the wrong origin. Upon receiving the notification, the NOFRAK JavaScript library retrieves the data via a Web-to-local bridge authenticated by the token.

The other mechanism is *active polling*. After submitting a local request, the NOFRAK JavaScript library keeps polling the local half for a response. Each query is authenticated by the token.

Designing the local-to-Web bridge would have been easier if the local half of NOFRAK could write its responses directly into the localStorage associated with the origin that made the request. Unfortunately, existing embedded browsers do not have an API for doing this securely, and redesigning the browser would have greatly impaired the portability of NOFRAK.

In addition to authenticating bridge invocations, NOFRAK authenticates requests to change the bridge mode made from the Web side in exactly the same way. Therefore, NOFRAK supports future additions to the set of available bridges and prevents chosen-bridge attacks by foreign-origin content.

C. Evaluation

To demonstrate that our implementation of NOFRAK is transparent to all legitimate hybrid apps, it is sufficient to verify that the API of NOFRAK-enhanced PhoneGap is indistinguishable from the API of stock PhoneGap. Because legitimate apps interact with PhoneGap only via this API, this criterion obviates the need to test individual apps.

To this purpose, we considered all public API functions of PhoneGap. As of this writing, the PhoneGap API has 16 categories: Accelerometer, Camera, Capture, Compass, Connection, Contacts, Device, Events, File, Geolocation, Globalization, InAppBrowser, Media, Notification, Splashscreen, and Storage. Table I shows the number of public API methods for each category. Connection, Events, and Device do not have any public methods. Connection uses 1 public property to retrieve the connection type, Device uses 6 public properties to fetch device information, and Events has 15 event handlers. We developed a JavaScript test suite and verified that in 100% of the tests, PhoneGap returns exactly the same results with and without NOFRAK.

To measure the performance overhead of NOFRAK, we benchmarked NOFRAK-enhanced PhoneGap against stock PhoneGap. Averaged over 10 independent executions, NOFRAK adds approximately 0.24% overhead to synchronous calls and 5.86% overhead to asynchronous calls as shown in in Table II.

Device Resource	**# of public methods**
Accelerometer	3
Camera	3
Capture	4
Compass	3
Contacts	5
File	36
Geolocation	3
Globalization	12
InAppBrowser	6
Media	10
Notification	5
Splashscreen	2
Storage	8

TABLE I: Number of public methods for accessing different device resources in PhoneGap

	PhoneGap	NOFRAK	**Overhead**
Sync	1.7713 ms	1.7755 ms	1.0024x
Async	0.1244 ms	0.1317 ms	1.0586x

TABLE II: Performance overhead of NOFRAK

IX. RELATED WORK

Luo et al. [17] observed that interfaces added to Android's WebView via 'addJavascriptInterface' can be accessed by any script regardless of its origin. PhoneGap contains defenses against this attack and also implements other bridges without this particular vulnerability. Luo et al. studied two applications but did not find any actual vulnerabilities since the applications in question do not expose any sensitive resources through 'addJavascriptInterface'.

In this paper, we carry out a comprehensive security analysis of the hybrid software stack and demonstrate that fracking vulnerabilities are generic and affect all bridge mechanisms, all embedded browsers, all hybrid frameworks, and all platforms. Many of these vulnerabilities (e.g., those in event-based bridges, which do preserve the origin of the call) not caused by frame confusion and thus different in nature from the 'addJavascriptInterface' vulnerability.

Luo et al. briefly dismiss authentication with pseudo-random tokens by arguing that sensitive data returned by the framework's local half may be mistakenly delivered to a malicious main frame. We are not aware of any hybrid app whose main frame has a different Web origin than the app itself. Typically, it is either a local file, or has the same origin as the app. In any case, our NOFRAK defense ensures that only the authorized origins can access the returned data even if the main frame is malicious.

Some mobile advertising libraries on Android expose device resources via 'addJavascriptInterface' to JavaScript ads running inside WebView [11, 26]. Stevens et al. [26] also found that some of these libraries fetch content over HTTP and are thus vulnerable to network attacks. Their proposed solution is privilege separation between the browser instance running the advertising library and the actual mobile app. As explained in Section VII-F, local privilege separation is much more difficult to apply to hybrid apps because—like conventional Web apps on which they are based—they "mash up" content from trusted and untrusted Web origins within the same browser instance.

Besides exposing device resources, mobile advertising libraries can cause over-privileging of mobile apps. Pearce et al. [21] added advertising API support and corresponding advertising permissions as part of the Android platform instead of running the entire advertising library as part of the application. AdSplit [24] enforces privilege separation by running the

library and applications as two separate processes with overlaid displays.

With additional system support, privilege separation can also help prevent click frauds. Quire [8] uses call chains and digital signatures to establish the provenance of IPC calls made by Android applications. It can be used to verify that apps correctly display ads and users indeed click on them.

Wang et al. [27] analyzed cross-origin vulnerabilities in inter-application communication channels such as intents, schemes, and Web-access utility classes on mobile platforms. Their threat model involves users installing malicious apps or clicking on malicious Web links. Morbs, the defense proposed in [27], is based on labeling all inter-application messages with their respective origins and enforcing origin-based security policies. In contrast, this paper focuses on a different part of the software stack, namely, device access by untrusted Web content running in embedded Web browsers, and demonstrates the need to apply origin-based access control there, too.

Davi et al. [7] analyzed permission re-delegation attacks on Android applications. Permission re-delegation is an instance of the confused deputy problem [14] where a privileged application exposes some operations as a service that non-privileged applications can invoke, yet does not correctly check the credentials of the invoking application. Felt et al. [9] found that many Android applications suffer from permission re-delegation attacks. Many defenses against re-delegation attacks have been proposed in the literature [6, 8, 9, 12, 16].

Fracking vulnerabilities described in this paper can be viewed as an instance of the confused deputy problem, but they occur at a different level of the software stack than the permission re-delegation attacks. Fracking vulnerabilities result from a mismatch between the security models of the app's Web code (governed by the same origin policy) and the framework's local code (governed by the platform's access control policy). Fracking vulnerabilities are not specific to Android and apply to all platforms and all mechanisms currently used in hybrid frameworks to expose local resources to Web code.

Chen et al. [5] proposed using permission event graphs to prevent malicious applications from misusing their privileges by enforcing OS-context-specific policies on them.

Proposals for finer-grained access control than the current Android system include byte-code rewriting [15], intercepting calls to Android native libraries [31], and modifying the Android OS itself [20]. Hao et al. [13] showed that incomplete implementation of such fine-grained access control using Java byte-code rewriting can be bypassed by malicious applications. Fine-grained access control at the OS level does not help against fracking attacks if the OS cannot distinguish whether a particular access request came from trusted or untrusted Web content within the browser.

Security vulnerabilities are often caused by the application developer's misunderstanding of an SDK or framework API [10, 28]. Fracking vulnerabilities occur in the hybrid framework itself and are not caused by the developers' misuse of the framework's API.

X. Conclusions

Hybrid applications—and, in general, various mechanisms for opening the browser sandbox and adding native access to Web code—are here to stay. Today, hybrid applications are typically built using one of the application development frameworks analyzed in this paper, but similar functionality is being added to conventional Web browsers, too. Chrome and the nightly versions of Firefox support the WebRTC standard that enables Web content to access local audio and video resources [29]. Chrome also supports "packaged apps"[12] with native access capabilities, Intel's MobiUS app browser[13] supports access to device resources and the user's social media, etc.

Hybrid software will continue to present security challenges. Existing app stores rely on semi-automated static audit to filter out malicious apps, but hybrid apps compose their content dynamically. This was less of a problem in conventional Web applications because they did not have any local access, but hybrid apps do. App platforms must develop dynamic, runtime mechanisms for recognizing and blocking malicious behavior.

Hybrid apps contain complex interactions between Web code and local code. Unfortunately, the Web security model and the local security model are not coherent, and the loss of origin when Web content accesses local resources can be devastating. Furthermore, even minor bugs in either the hybrid code, or the embedded browser open the door to cross-site scripting attacks.[14]

Domain whitelisting is now done opaquely by app creators. Showing the whitelists to the user may help the user make more educated decisions about (not) installing certain apps.

Security of the hybrid software stack is a complex, poorly understood topic that will only grow in importance. We view this paper as a step towards better understanding of the issues and designing robust defenses.

Acknowledgments. This work was partially supported by the NSF grants CNS-0746888, CNS-0905602, and CNS-1223396, a Google research award, the MURI program under AFOSR Grant No. FA9550-08-1-0352, NIH grant R01 LM011028-01 from the National Library of Medicine, and Google PhD Fellowship to Suman Jana.

References

[1] Abusing WebView JavaScript bridges. http://50.56.33.56/blog/?p=314.

[2] A. Barth. The Web origin concept. http://tools.ietf.org/html/rfc6454.

[3] BlackBerry 101 - Application permissions. http://crackberry.com/blackberry-101-application-permissions.

[4] HTML5/WebWorks for BB OS, BB10 and PlayBook. https://developer.blackberry.com/html5/.

[5] K. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song. Contextual

[12] http://developer.chrome.com/apps/about_apps.html

[13] http://dev.html5m.com/?q=mobius

[14] https://github.com/blackberry/BB10-WebWorks-Framework/issues/82

policy enforcement in Android applications with permission event graphs. In *NDSS*, 2013.

[6] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *MobiSys*, 2011.

[7] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *ISC*, 2010.

[8] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security*, 2011.

[9] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

[10] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *CCS*, 2012.

[11] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.

[12] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, 2012.

[13] H. Hao, V. Singh, and W. Du. On the effectiveness of API-level access control using bytecode rewriting in Android. In *ASIACCS*, 2013.

[14] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

[15] J. Jeon, K. Micinski, J. Vaughan, N. Reddy, Y. Zhu, J. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android. In *SPSM*, 2011.

[16] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *CCS*, 2012.

[17] T. Luo, H. Hao, W. Du, Y. Wang, and Y. Heng. Attacks on WebView in the Android system. In *ACSAC*, 2011.

[18] WebView addJavascriptInterface remote code execution. https://labs.mwrinfosecurity.com/blog/2013/09/24/webview-addjavascriptinterface-remote-code-execution/.

[19] Building Android Java/JavaScript Bridges. http://labs.mwrinfosecurity.com/blog/2012/04/30/building-android-javajavascript-bridges/.

[20] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In *ASIACCS*, 2010.

[21] P. Pearce, A. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege separation for applications and advertisers in Android. In *ASIACCS*, 2012.

[22] Domain Whitelist Guide. http://docs.phonegap.com/en/2.6.0/guide_whitelist_index.md.html.

[23] E. Shapira. Analyzing an Android WebView exploit. http://blogs.avg.com/mobile/analyzing-android-webview-exploit/.

[24] S. Shekhar, M. Dietz, and D. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security*, 2012.

[25] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postMessage in HTML5 websites. In *NDSS*, 2013.

[26] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in Android ad libraries. In *MoST*, 2012.

[27] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *CCS*, 2013.

[28] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security*, 2013.

[29] WebRTC native APIs. http://www.webrtc.org/reference/native-apis.

[30] App capabilities and hardware requirements for Windows Phone. http://msdn.microsoft.com/en-us/library/windowsphone/develop/jj206936(v=vs.105).aspx.

[31] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security*, 2012.