The Demon is in the Configuration: Revisiting Hybrid Mobile Apps Configuration Model

Abeer AlJarrah
University of North Carolina at Charlotte
aaljarra@uncc.edu

Mohamed Shehab
University of North Carolina at Charlotte
mshehab@uncc.edu

ABSTRACT

HTML-5 hybrid apps have the potential to dominate the mobile and IoTs market as hybrid platforms are providing a promising development choice. This approach "wraps" standard web code (HTML, Javascript, and CSS) into a thin native layer, enabling the same code base to run on several platforms. This approach also provides a mechanism to access device native sensors, such as camera and geolocation, through Javascript code. Apache Cordova is an open source library that is a common component in many hybrid platforms, such as PhoneGap and IBM Worklight. Yet, its configuration model suffers several security limitations including a coarse-grained access control model, risky defaults, and for many developers, a non-trivial configuration process. Hybrid app development is an intricate task as is, not to mention configuring these apps securely. Given the increased popularity of the approach itself and the proven tendency of developers to use platform-provided default settings, this paper presents a novel approach to automatically generate configurations that are more aligned to the app requirements, more granular, and more conformant with Least Privilege principle. We argue that having aligned configurations forms the first line of defense against attacks similar to injection attacks. Such attacks could have been voided if the app configurations were more granular and tailored. Our approach generates initial configuration settings based on modeling app behavior. The model generates twofold policies, one centered around APIs access and another around controlling app states transition. We have successfully instrumented Cordova library to implement our approach. We have tested the instrumented version, and our experiments demonstrate that the instrumented version is a practical and performant alternative.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; *Web application security*; *Domain-specific security and privacy architectures*;

KEYWORDS

Mobile Hybrid Platforms Security, Configurations, Cordova, PhoneGap, HTML5, Application Security

ARES '17, August 29-September 01, 2017, Reggio Calabria, Italy

ACM ISBN 978-1-4503-5257-4/17/08…$15.00
https://doi.org/10.1145/3098954.3105825

ACM Reference format:
Abeer AlJarrah and Mohamed Shehab. 2017. The Demon is in the Configuration: Revisiting Hybrid Mobile Apps Configuration Model. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 10 pages.
https://doi.org/10.1145/3098954.3105825

1 INTRODUCTION

Smartphones, wearable devices, and the system of Internet of Things (IoTs) are eventually becoming mainstream, replacing desktops not only as personal gadgets but also as workplace tools. This will increase the demand for Enterprise mobile apps, which is expected to outstrip available development capacity 5 to 1 by the end of 2017 according to Gartner[37]. Enterprises are striving to maintain apps that can run on different devices with low cost. This explains enterprises' increasing interest in adopting Mobile Hybrid development approach [28] because this approach satisfies the business need to leverage mobile applications across many platforms. It also provides the capability to use mobile device features using standard web technology. Adopting this approach eases administrative tasks in the Bring Your Own Device (BYOD) environment. Moreover, it enables using single code base on many platforms, which drastically reduces the cost by targeting the whole market and discarding platforms' market fragmentation [4][41]. Mobile Hybrid development uses standard web technology (HTML5, JavaScript, CSS) "wrapped" into a thin native layer, which enables the app to run on different platforms. This approach is a mix of the other two approaches on mobile development, namely *Native* and *Web-Based*. Native apps' advantage of accessing device native features is leveraged in Hybrid apps through the native wrapper. At the same time, the same code base is used for different platforms, making it possible to ship to a wider customer base in a low cost.

As Hybrid approach continues to experience acceptance within the developers' community, organizations must think seriously about how key changes in this latest paradigm will require them to shift their application security practices for Web and Mobile Apps. The effect of any attack on a hybrid app is amplified because the app has the ability to access device native features through a bridge implemented by the hybrid platform middleware [24][25].

Cordova Library [1] is a middle-ware that is a common component in many popular hybrid platforms , such as PhoneGap, IBM Worklight, App Builder, Sencha, Monaca, and Appery.io. This component is the real enabler of connecting the two worlds (Web and Native) inside a hybrid app. However, the library endures limitations of unsafe defaults, and a coarse-grained configuration model, which raises security concerns. Moreover, the library has vague configuration documentation especially those related to security settings [3].

In the latest release of the ten most critical web application security risks, OWASP [8] indicates that security misconfiguration is a serious issue given that defaults are usually not secure. This is congruent with the results of scanning 662 Cordova-based apps [34], which uncovered a serious issue in that regard. We have found that 81% of the apps have misaligned configurations, meaning they have API declarations that are not actually used. Moreover, 58% have risky settings such as allowing loading resources from *any* domain. This is a result of using the default settings provided by the library, which is not necessarily secure, especially, the early versions of Cordova library[1] that do not provide a default Content Security Policy (CSP). Not to mention, developers' tendency to overlook security settings and focus only on the functional part of the app.

This work aims to enhance Cordova library configuration model by suggesting a fine-grained configurations scheme and also generating an initial set of configuration values elicited from the app behavior. Providing fine-grained and aligned configurations not only protects the app against potential attacks, but also adds minimal effort on the developer's side, by automatically generating an initial set of fine-grained configuration values to govern the app behavior in terms of plugin access and state transition. The proposed approach captures rules of state-plugin access rules and state transitions by monitoring the app behavior while running in a controlled environment and then enforcing these rules when the app is released to the market. The goals of this approach are:

(1) Suggest a more fine-grained configuration scheme for hybrid apps that is based on app state level rather than the current global app level model.
(2) Provide the developer with initial configuration values based on the app behavior

Research effort focusing on hybrid app security has been dealing with this matter in a sequential and a reactive course [24][27][19]. So far, attack channels are discovered first, and then in response, a change to the platform is proposed to mitigate or prevent these attacks, which makes securing hybrid apps a slow process. Our approach is addressing risks early by focusing on having aligned configurations rather than default configurations. The discussion of this work is mainly about Cordova-based apps on Android. However, concerns, implications, and solutions are still applicable for other platforms. Our main contributions are:

- We propose, implement, and test a behavior-based approach for app configuration, that requires minimal effort from the developer's side, yet helps in configuring hybrid apps securely by:
  - Generating an aligned, fine-grained policy-centric to device plugin access.
  - Generating a state transition verification model to prevent and detect potential exploits targeting app behavior.
- We demonstrate the risk and the method of targeting and manipulating app behavior.

The paper is organized as follows, section 2 provides a background on Cordova library, HTML-5 apps development, and their security. Section 3 discusses risky configurations and their implications. After that, in section 4 , we present our approach, explaining the process of modeling the app behavior and the mechanism to enforce it through secure configurations. Next, we review related work in Section 5. Finally, we conclude and discuss future work in Section 6.

2 BACKGROUND

2.1 Apache Cordova Library

An open source library that enables the creation of mobile apps with HTML, CSS, and JS. It targets multiple platforms with one code base. It supports 8 platforms including Android, iOS, Windows, and Blackberry. Applications execute within "wrappers" targeted to each platform and rely on standard-compliant API bindings to access each device's sensors, data, and network status[1]. Diverse platforms use this library, including PhoneGap, Ionic, Visual Studio and Intel XDK [16]. Despite its' relatively recent presence, Cordova-based apps recent statistics in Google Play shows that it constitutes 5.84% of the market share. Some apps are able to attract a customer base of 10,000,000+ [5]. Business, Medical, and Finance tops the list of Cordova-based app categories. Moreover, observing the platform's increasing popularity, recent research suggested augmenting Cordova library with wider functionalities, such as voice agent and embedded WebRTC [36][23].

Cordova applications are implemented as a browser-based `WebView` within the native mobile platform. A Cordova plugin is an add-on code that provides JavaScript interface to native components. They allow the app to use native device capabilities beyond what is available to pure web apps such as camera, contacts and geolocation. At the time of writing this paper, Cordova supports 1069 plugins that are open source and available to the public[1].

Cordova library implements a bridge to connect the two worlds (Web and Native) together. `CordovaWebView` adds a Javascript Interface using the method `AddJavascriptInterface()`, which enables the WebView's internal Javascript code to call the native method `CordovaPlugin.exec()`. The `exec()` function is the entry point to any plugin on the native side of the app. After the native side executes the plugin, the result is saved in a queue of JS messages that are injected back to the WebView using `loadUrl()`. Moreover, the Cordova JS side might trigger the WebView to display an alert dialog, confirmation or prompt. This is implemented by customizing the event handlers in `WebChromeClient`. This object mainly models how WebView should react to Javascript dialogs, favicons, titles, and the progress through overriding methods.

A Cordova-based app, regardless of the platform uses one global configuration source file namely `config.xml`. This XML based file located under `/res/xml` path and contains several settings that control app behavior. The configurations are mainly updated through Cordova command line interface (CLI) or by simply editing the file. Default configurations specify global properties, such as name, description, author, preferences and domain whitelisting specifications. Adding a plugin API changes the configuration to include a declaration of that plugin by adding a `<feature>` tag. CLI also adds the required system permissions to the native side to permit using the device native features. The default `config.xml`, according to the latest version (*version 6.x*), only includes the `Whitelist` feature. This feature manifests the security model by providing the developer with configuration items that control the app interactions

Syntax	Default	Meaning
`access origin="*"`	✓	Allow accessing resources from all domains
`allow-navigation href="http(s)://*/*"`		Allow links to all http(s) URLs to be loaded
`allow-navigation href="data:*"`		Allow data of all formats to be passed into the WebView
`allow-intent href="http(s)://*/*"`	✓	Allow all http(s) links to web pages to open in a browser
`allow-intent href="sms:*"`	✓	Allow SMS links to open messaging app
`allow-intent href="geo:*"`	✓	Allow geo: links to open maps
`allow-intent href="tel:*"`	✓	Allow tel: links to open the dialer
`allow-intent href="market:*"`	✓	Allow market: links to open Google Play Store

Table 1: Whitelist Configurations

with external entities, such as URLs and other installed apps, using domain whitelisting model. Table 1 explains the specifications provided by this plugin indicating the defaults. Domain whitelist configurations can be categorized into:
- Network Request Whitelisting
- Navigation Whitelisting
- Intent Whitelisting.

Network Request Whitelisting defines the set of external domains the app is allowed to communicate with, specified by `<access origin= "*"/>` by default, which allows the app to accept resource access from ***any*** server. Navigation Whitelisting controls which URLs the `WebView` itself can be navigated to. By default, navigation is allowed only to `file://`, which means the app is allowed to navigate only to local HTML files. If developers want to add other external URLs, she must add `<allow-navigation href=URL/>`. Intent Whitelisting controls interaction with installed apps, such as the browser, dialer, messaging and built-in maps. Using this specification also allows passing parameters to the intent, such as a specific URL. Recently (as of *version 4.0* [3]), a default Content Security Policy (CSP) is included in the startup *index.html* generated by the library. According to the library documentation, CSP is meant to control network requests, such as images and XML HTTP Requests (XHRs), made via WebView directly that can not be controlled using Network Request Whitelisting. On Android, for instance, the network request whitelist is not able to filter all types of requests, such as `<video>` and WebSockets. So, in addition to the whitelist, a CSP via `<meta>` tag is required on all pages. However, CSP is not effective nor practical for two main reasons. First, setting a CSP is ***not*** mandatory in Cordova-apps. Meaning, if a developer copied her web-based files into the `www` folder inside the app replacing the default *index.html*, there will be no CSP, yet the app runs with no errors. Second, CSP documentation is neither simple nor informative in terms of potential risks of not setting CSP properly. This may not encourage developers to use it, especially those whose ultimate priority is getting the app to work. Such developers are not expected to go further into considering security policies in their code.

2.2 Mobile Hybrid Apps Security

Mobile Hybrid Apps introduce a new software paradigm, which also induces new challenges to security. We explain common attack vectors and threats specific to this category.

I. Code Injection Attacks: Given that JS is subject to code injection and that hybrid apps use standard web technology, attackers can abuse smart phone specific features, such as, camera as a new window for code injection. The essence of this model stems from two basic facts:

- Data and code can be mixed and the mixed code can be triggered by standard JS engine inside `CordovaWebView`.
- Smart phones, by nature, provide broader interaction surface with the outer world compared to PCs, through device native features, such as camera.

Cordova input plugins can be abused as channels to inject infected payloads of data into the app [24]. Untrusted data input resources, such as WiFi access points, Bluetooth, Bar-codes, QR images and MP3 files, can embed malicious code that can be triggered inside the infected app through the JS engine.

Recent work [22] has also discussed Web-to-Application injection attacks. Where malicious code injection can happen by passing malicious data to `Intent` through a link call. The target of such an attack is the installed apps on the device. Injected code may abuse the bridge enabled to launch installed apps, such as dialer and maps. One compromise scenario would be invoking a `BROWSABLE` intent passing malicious `url` as a parameter.

II. Compromised Third-Party Providers: An important feature of JavaScript is the ability to combine many libraries from local and remote resources into the same page. Developers include remote resources for many reasons, such as decreased latency, increased parallelism, and better caching. Mobile Hybrid Apps developers in particular strive to create cross platform apps with a native look, high performance and rich functionality. Thus, UI specialized libraries (jQuery, Ionic, Framework7, Mobile Angular UI), Tracking libraries (Google Analytics), Social Integration libraries (Facebook) or Ad libraries (AdMob), are very common in cross platform apps. Ideally, developers should include JS code from trustworthy providers, yet even these providers are not immune to attacks.

Although developers trust that remote providers will not abuse the power bestowed upon them, the amount of damage that can be caused by compromising these remote servers is substantial. Remote JS code has the same privilege as local code. Thus, it can access app resources such as data and device-native features.

2.3 Malicious Impact on Hybrid Apps

Malicious code damage can take several forms, such as data manipulation, privacy leakage, DoS, or changing app behavior. The impact of the attack and the way the attack is attempted differ based on the target of the attack. We categorize attacks on hybrid apps based on their target:

Device/User Data: Hybrid platforms provide a bridge to connect

```
<script>
  var refreshID = setInterval(function(){
  navigator.geolocation.watchPosition(
  function(loc) {
      $.post('someurl',format(loc),function(){});
      });}, 3000);
  function format(loc){
      var data= {'lat':loc.coords.latitude,
                 'long':loc.coords.longitude} ;
       return JSON.stringify(data);
  }
</script>
```

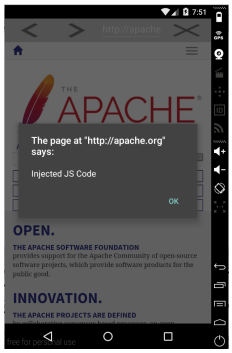
Figure 1: Targeting Device Data

to device sensors, thus malicious JS code may target the device sensors accessing private data, such as, tracking the location of a specific device as shown in Figure 1. The malicious code in this example posts the compromised device geolocation coordinates every 3 seconds to a remote server.

App Behavior: A recent market scan showed that finance, banking and health top the market share of Cordova based apps [5]. Thus, compromising the app behavior is a valid concern, particularly in these sectors. This can be achieved by malicious actions, such as, malicious redirect, manipulating the routing logic to bypass a mandatory check such as log-in or payment page, and injecting un-wanted content similar to advertising and adult content. The Cordova `InAppbrowser` plugin provides the functionality of redirect from one URL to another. By injecting code as the one shown in Figure **??** , the app redirects to another URL. The parameter `_self` means that the external URL (someurl) is hosted inside the app rather than the system browser. Moreover, this plugin enables JS code execution using the method `executeScript()`. Although, this JS code may not access Cordova APIs, it can manipulate the app behavior. The example shown in Figure 2 demonstrates how the code can successfully run. Default configurations (4th row in Table 1) allows ***any*** URL to load.

```
<script>
  var ref = cordova.InAppBrowser.open('http://www.someurl.com',
   '_self');
   ref.addEventListener('loadstop', function(){
   ref.executeScript({code:"alert('Injected JS Code')"});
   });
</script>
```

(a) Injecting JS Code using `executeScript()`



(b) App Behavior Change

Figure 2: Tampering App Behavior

2.4 HTML-5 Based App Development

HTML-5, Javascript and CSS have opened the door for platform independent UI development because it provides a rich UI experience not to mention support for several mobility features, such as offline web storage, canvas drawing, and CSS3. Considerations specific to mobile development must be taken into account to help provide the user with a full experience of a mobile app with good performance, given the limited processing capacity of mobile devices. Bandwidth is a major concern; thus, minimizing server requests and payload size, while providing rich UI, is a priority. To achieve this, mobile HTML-5 based apps implement most of the logic on the client-side, leaving server side for authentication and data access. This trend mandates the application of strict mechanisms to keep client side Javascript as layered, modular, and object-oriented as possible to help manage and maintain the code.

Single Page Applications, or (*SPA*), is a web-based app that fits into single page with the goal of providing more fluid user experience, akin to desktop application. Developers are highly encouraged to follow this approach when developing mobile apps for several reasons, such as improved performance, fast navigation between many views/pages, and less download content/network bandwidth in order to achieve more native like experience.

SPA attempts to reduce the total number of pages that a user must load to one. JavaScript routers provide a method for tracking user state and loading required resources, as needed, without requiring a URL change or page reload using *Hashbangs*. As the user navigates, the library changes the hashbang (#!) in the URL to denote their current location. Hashbangs are usually an alphanumeric word that represents certain action. It might also contain a parameter that can be a digit (for instance index in a list or an array) or a letter. Examples of hashbangs look like #employee/1/changePhoto, #/users/list/5, #pages/about. Gmail, Twitter, and Facebook are examples of SPAs. The urge to adopt this approach for hybrid apps is increasing. This also complies with our apps scan result, as approximately 60% of the apps pool have one HTML file.

3 "BAD" CONFIGURATIONS RISKS ON HYBRID APPS

App configuration is a contract between the app and the system. The file `config.xml` content states what external domains can access the app, what external apps can be launched from the app , what plugins the app can use, and what domains can be navigated from *inside* the app's `WebView`. Secure configurations serve as a first defense line against potential attacks. Previous work on securing hybrid apps [24][9][7] have been focusing only on Javascript (JS) as a source of problem and solution. The approaches suggested so far are revolving around static and dynamic JS analysis to detect malicious code injection. Configurations, however, have been missing from the literature even though it is a basic component of the app security. The damage from any malicious code execution can be voided by properly configuring the app. For example, an injection attack trying to access a camera is ineffective if the app is not configured to include the camera plugin in the first place.

As previously shown, default configurations in Table 1 are loose and coarse-grained. We demonstrate the impact of *bad* configurations that may result from either keeping default settings or having

relaxed configurations:

Network Request Whitelist: Controls which network requests are allowed to be made via Cordova native hooks. Default setting is `<access origin="*" />` that does not block any access request. Default index page generated by the library contains Content Security Policy (CSP) to control which network requests are directly allowed to be made via `WebView`. Default CSP allow only local URLs. CSP is enforced through the white-list plugin, which is added to the app by default. CSP is represented inside `<meta>` tags on the top of the HTML file. While including CSP is a secure practice that is encouraged by the library, not including one results only in showing a warning message. Controlling network requests is merely dependent on the possibility of the developer using default generated index page that contains a default CSP , or the possibility of her adding a CSP to her customized pages. Having no CSP enforced and keeping default configurations may jeopardize the app to external resource loading/injection from *any* URL (1st row in Table 1).

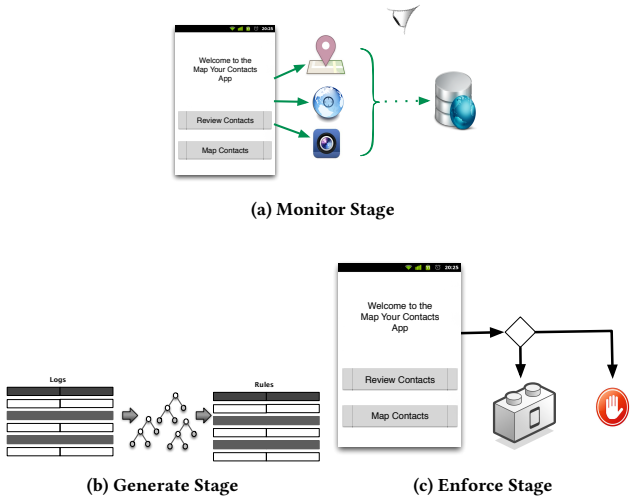
Navigation Whitelist: Controls which URLs the `WebView` itslef can be navigated to. This setting applies to top-level navigations only. By default, it navigates to local files because, as shown in rows 2,3 in Table 1, these settings are not default. To white-list specific URLs, configuration should include the corresponding value, for example, `allow-navigation href="http://example.com"`. However, as mentioned earlier, this applies only to top-level navigation, which means that any redirect from `example.com` to any other URL may not be in conformance with the specified setting. We argue the need of additional configuration settings to control all URLs being loaded into `WebView`. This helps not only protecting against malicious URLs loading, but also maintain the app behavior against any injection attack trying to bypass certain states in the app, such as authentication.

Intents Whitelist: Controls which URLs the app is allowed to ask the system to open (through hyperlinks and `window.open()`) . This includes Built-in Apps, such as, Messaging, Dialler, Maps, Mail, and Browser. Default settings allow calling *all* the mentioned apps as shown in rows 4-9 in Table 1. Including these apps in the configuration file allows malicious injected code to launch these apps, not to mention passing values to them using crafted URIs. In fact, Cordova versions before 3.5 suffered from a vulnerability (CVE-2014-3502) [10] that allows remote attackers to open and send data to arbitrary applications via a URL with a crafted URI scheme for an Android intent. Hence, we argue that keeping the default settings as is will increase the attack surface through this channel.

Declaring Plugins/Features: Early versions of Cordova configuration (before 3.1), used to include ,by default, a set of 16 declarations of native APIs or plugins, including Camera, Geolocation, and Contacts. This default setting assumes that the app needs to use all the declared plugins, which is not necessarily true. Previous work [34] demonstrated the gap between the plugin declaration in the configuration file and the actual plugin usage in the code. Even if a developer included only the plugins needed by the app, these declarations assume that these plugins are used by every page/state in the app, which also creates a window for attack. We argue that plugin declaration should be tied with the app state/page that needs to access the plugin rather than being tied to the whole app as one unit.

Most developers consider security a non functional requirement. Thus, they are less likely to change default configurations. Even for developers who consider themselves security-sensitive, there is still a possibility that they might lack the knowledge to do so. This outlook is backed by a study [42] that reveals the existence of a disconnect between developers' conceptual understanding of security and their attitudes regarding their personal responsibility and practices for software security.

4 BEHAVIOR-BASED SECURITY MODEL DESIGN



(a) Monitor Stage

(b) Generate Stage

(c) Enforce Stage

Figure 3: Three-Stage Behavior Policy

Configuring any application requires a profound understanding of its behavior. Static and dynamic code analysis approaches combined provide a comprehensive methodology of modeling not only application behavior but also other details such as data flow and control flows. Static analysis of cordova-based apps have been done in previous research [6]. While this approach may provide an accurate call graph with respect to static behavior in terms of cross language calls, dynamic app features may not be present in this analysis. Mobile Hybrid Apps are dynamic by nature. Changing app logic, especially through remote Javascript calls is reasonable. Moreover, API calls using obfuscated code may not be captured as well. Hence, we present a dynamic behavior modeling.

Our proposed approach captures *(state,plugin)* access rules by monitoring the app behavior while running in a controlled environment. It also captures the *states transitions* sequence and then generates a policy to control app states transitions. Our ultimate goal is to control malicious code's impact on the device and the app using more aligned and fine-grained configurations. We have chosen to monitor behavior in the stage of testing for two main reasons:

- We can control code coverage by using extensive testing scenarios that cover all API calls. Recommended App Testing practices requires testers to go through all functional

scenarios of the app, covering every possible functionality needed to be implemented. Thus, monitoring app behavior at this stage provides rich information about how the app should perform.
- Using an existing stage to monitor the app behavior comes in congruent with our goal of keeping the process as seamless as possible to the developer. Also, to support addressing potential vulnerabilities as early as the development and testing stage.

Our approach is focused on protecting the app against attacks that changes the app behavior, aiming to abuse the library implementation through accessing device features or tampering app behavior. If this malicious behavior is detected, then the code will be prevented from execution by setting configuration that defines app healthy behavior.

4.1 Plugin Access Policy

Our approach enables capturing app behavior and then enforcing access rules elicited from the behavior of the app. The approach is a three stage solution. The app should go through the *Monitor* stage, *Generate* stage and then *Enforce* stage as shown in Figure. 3. We extend the `config.xml` file to enable the management of tracking these stages by including a "stage" element. The Cordova library contains a function called `exec`, which serves as a central hub for plugin access. No plugin can be accessed without being passed to this function. Hence, we add our reference monitor logic in the `exec` function.

Monitor Stage: Occures when the app is running in a controlled environment, namely, while development and testing. Any access request to native plugins is monitored and associated with the app's current state. Here, "State" refers to the URL parts that include protocol, domain, subdomain path, fragment and parameters. This stage also captures the operation requested. Current state of the app could end with a page name, or a hashbang. Hashbang is anything after "#" tag in the URL which can be either #/action or #action[39]. For either single or multi page apps, all the URL parts are used to denote a state of the app. As described earlier, each `CordovaActivity` contains a `CordovaWebview` that extends the `Webview` class. We extract the current state by calling the `Webview` instance's `getUrl()` method. The information extracted are saved as Logs in a database.

Generate Stage This is an intermediate stage that is necessary to generate a behavior-based policy for the app. It is meant to be when the developer is done with development/testing and decides to release the app to the market. This stage is done once, when the app stage in `config.xml` is changed from *Monitor* to *Enforce*. This change triggers the process of reading logs that was previously saved during *Monitor* stage to extract a fine-grained policy that represents app behavior in terms of state-plugin rules. Processing the logs passes through the following steps:

I Build Call Map: This step reads all plugin access instances and groups them by plugin. As a result, a map of plugin, and a set of URLs are generated. This map (`callMap`) helps construct access rules per plugin in the second step.

II Build Call Tree: This stage is essential to group all app URLs into states through a syntax tree that represents all captured URLs of a specific plugin. For each plugin a syntax tree (`PluginCallTree`) is generated to represent the states such that it can be traversed later to generate a set of expressions that can be used for enforcement. The pair (plugin,`PluginCallTree`) is saved into another map `callTreeMap`.

III Extract Patterns: This step traverses the `callTreeMap` to extract regular expressions for every plugin. The list of regular expressions will be used to form the access policy used by the third stage *enforce()*.

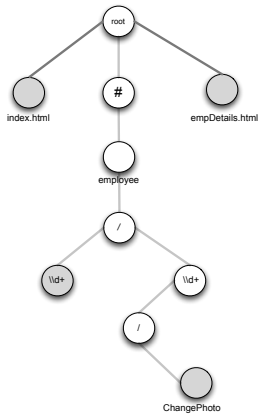


Figure 4: Abstraction of Syntax Tree Representing States

A simple heuristic function is implemented for step II that is responsible of identifying app states. For multi-page apps, URLs can be used as is to serve as identifiers for the app states. In single-page apps (SPA) however, URL fragments are considered. Generally, SPA URLs follow the following scheme:

$$\#[/]action[[/index]^*[/action]^*]^*$$

Where action is an alphanumeric word that normally describes a function. Index is either a digit or character that is used to identify an instance of data. For example, if a plugin can have these URLs captured: index.html, #/employee/5/, #/employee/10/, #/employee/6/, #/employee/12/ChangePhoto and empDetails.html. The syntax tree that will be generated to represent all states modeled by the tree on the left in Figure 4. To minimize the tree size and to avoid the need of representing every single possible value of an index, the heuristic function combines and abstracts nodes that represent indexes accessing the same plugin. If the the heuristic found repeated nodes (more than a specific threshold) that differ only in the index, then it infers that there is a pattern of calling the plugin. To represent the pattern, the nodes that represent indexes are replaced with a generic expression that represents the properties of the repeated indexes. Our implementation uses Java; so we used Java regular expressions to represent states. Index in number or character are replaced by "d+" or "w" so that when the states are traversed back, we get Java regular expressions to represent integer-indexed view state; for example, "#/employee/d+" represents parameterized view state #/employee in this scenario. The left tree in Figure 4 shows the resulted tree if threshold is set to 3. Similar URLs are grouped as an expression. Extracted expressions can be used in the policy to validate subjects.

Enforce Stage This stage is meant to be when the app is released to

the market. A fine-grained policy on state level is generated based on the behavior that has been captured . Figure 5 shows a sample

```
<feature name="Storage" >
    <param name="android-package" value=".." />
</feature>
<feature name="Camera">
    <param name="android-package" value=".." />
    <param name="state" value="#/employee/\\d+" />
</feature>
<feature name="Contacts">
    <param name="android-package" value=".." />
    <param name="state" value="#/employee/\\d+" />
    <param name="state" value="empDetail.html" />
</feature>
```

Figure 5: Policy Rules in `config.xml`

policy that can be mapped to a simple Access Control Rule syntax composed of subject:(who can access) and object:(what resources to be accessed). In this case, subjects are the "states" generated and objects are the plugins. The policy is represented by a parameter attached to each plugin declaration in the configuration file. Plugins without any `state` parameter can not be accessed through any state of the app. Plugins added by default or by mistake, but are not actually needed in the app will have no state to be accessed from which will void the effect of including them. However, plugins that have parameter attached to it will be executed in the states that conforms to the value(s) indicated. For instance, in Figure (5) Camera plugin is only allowed to be accessed in a URL that conforms to the regular expression "#/employee/d+" where Contacts plugin can be also accessed from another page "empDetail.html". Storage on the other hand, cannot be accessed at all because it has not been captured by the reference monitor. The instrumented library generates this policy and updates the `config.xml` file of the app. The function *enforce()* that is added in the `exec()` function will use these rules to verify access after decrypting the state values using assigned private key. If access request complies with any rule, access is granted; otherwise a `PluginException` is raised and user notification of an invalid access appears.

4.2 Behavior State Modeling

We propose an approach that enforces controlling transitions between different app screens based on the app behavior. The essence of the approach is very similar to the previous section in terms of monitoring app state transitions, creating a policy and then enforcing behavior to prevent any potential app logic manipulation. A behavior is also defined by app interactions with other apps, such as, Dialer, Messaging and Maps. It is possible to call other apps the same way a link to a web page is called through using `<a href>` tag. For example `<a href="tel:+1-800-555-1234">call this number</a>`, enables the user to open the Dialer and pass a specific number when the link is clicked.

Similar to the process of app behavior elicitation approach discussed earlier, this is also a three stage solution. The app should go through *Monitor* stage, *Generate* stage, then *Enforce* stage.

Monitor Stage: At this stage, any URL load change, DOM view transition, or external app call are monitored and saved. For single page DOM management, we extended the "router" function on JS side to extract current `window.location` value and log it. This log is handled in the native class `SystemWebChromeClient` where the URL is then saved to a database (DB) through the native side.

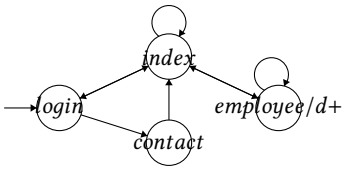
URL transitions to another page can happen either by navigating to another domain or loading the URL into the `CordovaWebView`. Capturing this transition can be done by monitoring `PluginManager.exec()` for the plugin `InAppBrowser`. We also monitor information such as URL and target host (*i.e.* WebView or System Browser).

App calls to other apps through crafted URIs starting with `tel:` or `sms`,for example, are monitored by adding a reference monitor in class `SystemWebViewClient`, specifically in the method `shouldOverrideUrlLoading` which gets triggered whenever a URL load happen. **Generate Stage** This intermediate stage is necessary to generate an XML representation of the state machine notation for control abstraction. The DB contains logs of URLs and page transitions in the sequence they were called. Then XML is generated to dictate the states the app have passed by.

As for external apps, no policy is required. Instead it is enough to generate navigations whitelisting only for those apps captured during the monitor stage. Figure. (6a) shows a sample state-policy generated by our sample app. The representation follows State Chart XML (SCXML) standard for representing the app control

```
<urlstate id='login' value='login.html' type=start
location=local>
   <redirect id='index'>
   <redirect id='contact'>
</urlstate>
<urlstate id='index' value='index.html'
location=local>
   <redirect id='index'>
   <redirect id='login'>
   <redirect id='employeeview'>
</urlstate>
<urlstate id='employeeview' value='#employee/d+'
location=local>
   <redirect id='employeeview'>
   <redirect id='index'>
</urlstate>
<urlstate id='contact' value='https://www.abc.com/
contactus' location=external>
   <redirect id='index'>
</urlstate>
<allow-intent href="tel:*" />
<allow-intent href="http://www.abc.com/" />
```

(a) App state machine XML representation



(b) App state diagram

```
<script>
  var oldHash = window.location.hash //global variable
  $(window).on('hashchange', $.proxy(this.checkRoute, this));
  checkRoute: function() {
      var hash = window.location.hash;
      var states = [oldHash, hash];
      if(navigator.Redirectlist.checkTransition(states,
      route(hash), route("error"));
      oldHash = hash ;
  }
</script>
```

(c) Check Redirection

Figure 6: App State Configurations

flow[40]. At this point, the implementation captures the state signified by <urlstate>. It also captures 3 other features:

- If it is a start state, represented by the value of type attribute.
- If it is an external or local URL state, represented by the value of location attribute
- All the redirects (transitions) from the state to other states.

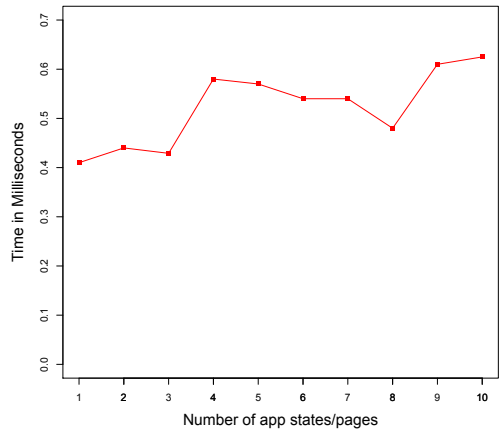
Developers need to copy this into their config.xml to be used for the *Enforce()* stage.

Enforce Stage A call graph is built based on the state machine configuration found in config.xml. Based on the example shown in Figure. (6a). Transition from one state to another is allowed only if the next state already exist as a destination from the current state, otherwise the app cancels redirection. Our Java implementation of this graph is done using a HashMap of states and each state has an ArrayList to hold the adjacency list to other states for the corresponding state. Any URL change will require either InAppBrowser to be called or onhashchange to be triggered; thus, we added the check in two places. For URL loading using the plugin InAppBrowser, we added the check in the PluginManager.exec() which is as it has been already mentioned, a central hub for all plugin calls. For internal DOM routing, we have developed a Cordova API to handle redirects (Redirectlist). This plugin is very similar to Cordova's built-in Whitelist plugin. The API parses the config.xml to create a HashMap to use for checking. We use the trigger window.onhashchange() to check change in state for SPAs as shown in Figure. (6c). To control the app interaction with external entities, the approach includes the navigation whitelisting settings. For example, last two lines of configurations shown in Figure (6a) includes only Dialer and Browser to specific URL, instead of including *all* built-in apps and *all* URL (see Table 1).

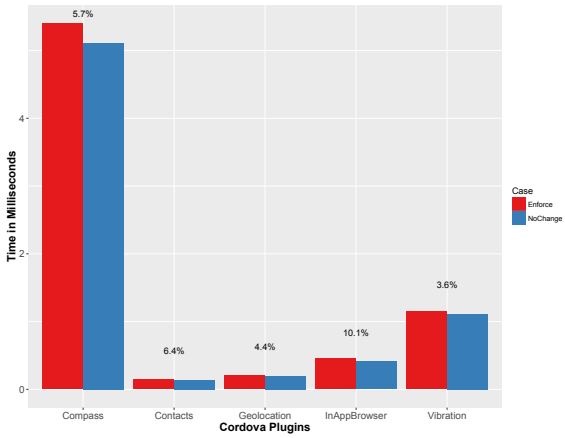
4.3 Performance Analysis

Performance overhead has been associated with dynamic code analysis as a disadvantage.This section describes performance measurements of different aspects of the proposed instrumented library compared to the stock version of Cordova library (*version 6.x*). We have used a device with the following specifications: Model Moto G (2nd Generation), Android *version 5.0.2*, Internal Storage 5.5 GB.

4.3.1 Enforce Time vs App Size. The purpose of this test is to measure the scalability factor of enforcing plugin access policy on varying number of pages/states of a hybrid app. For this test, we use one plugin (Compass) with 10 apps having 1 to 10 pages/states. We choose *Compass* plugin because it is a plugin that requires a sensor access and at the same time has a frequency option that we can use to enable automatic frequent calls from the code. We have also chosen to test up to 10 pages because the average ratio of page to app based on the apps pool [34] was 10 to 1. Hence, we conducted a series of tests measuring the time of enforcing a plugin access on 10 apps having 1 up to 10 pages/states. The time needed by checking the policy is the time between a plugin request and the decision to grant or prevent plugin access. This interval is measured for every single plugin access in every page/state and then the readings are averaged, Figure. (7a) shows that checking the policy varies between 0.4 to 0.58 milliseconds. The trend line shows that the time is linearly increasing with a positive slope value of 0.021. Although the app size and policy check time are positively proportional, the number of pages/states does not seem to be a strong factor of increasing the time given the low slope value.



(a) *Enforce* Time *vs* # States



(b) exec() *with* Plugin Type

Figure 7: Performance Testing

4.3.2 Original vs Modified exec() *Time with Plugin Type.* The purpose of this test is to check the relative overhead added by enforcing a policy related to a specific plugin and executing it compared to the time needed by the stock version. Another goal of this test is inspect how different plugin types perform in both cases. As Figure. (7b) indicates, a set of plugins selected that represent a variety of plugin categories. Compass, Geolocation, and Vibration are plugins that access the device sensors. While Contacts is a plugin that access the device contacts database and the InAppBrowser which does not access any device specific feature but is designed to host web content in a container other than the CordovaWebView. This plugin also is used in our new library for controlling external URL loads. The experiment shows that the time needed by Modified exec() (with *Enforce*) relative to the stock version varies between

3.6% for `Vibration` to 10.1% for `InAppBrowser`. The type of the plugin does not seem to affect the relative overhead time.

4.3.3 Redirect Plugin Check Time. As for internal DOM manipulation and URL redirects, we have measured the time it takes a router function to decide the view and render it in 2 cases, regular routing and routing after checking the `RedirectList` plugin. Averaging the time of 100+ runs of both cases revealed the following: Routing between views without checking takes only ≈ 10.8 microseconds while after checking `RedirectList` plugin takes ≈ 50.6 microseconds. The difference can be attributed to the round about time of messages between JS and Native side.

5 LITERATURE REVIEW

Cordova Library Access Control Model: Early work related to hybrid apps security [27][35][34][19] has focused on the coarse grained access control model offered by the library and on the risk associated with having a bridge implemented to connected to native device resources. Both works [27][35] suggested a finer-grained access model to control the native side permissions associated with external resources. MobileIFC [35] proposed to include a context-aware policy to control access to resources on both sides, native and web. The work suggested splitting JS code into confined chunks that are activated based on certain conditions. While Jin et al [27] have discussed how the privilege granted to Cordova-based Webviews breaks the existing protection mechanism provided by Android Webviews. They have proposed a fine-grained access control model on the *frame* level to be associated with native side permissions. The limitation of the solution offered by both, is that it imposes usability requirements and responsibility on the user and developer side. Both solutions are as effective as the user/developer are concerned and willing to set security policies. On the other hand, Shehab and AlJarrah [34] have suggested to associate Cordova API access to page level to mitigate attack effect on multi-page apps. However, with the increasing trend toward Single Page Apps; the solution does not provide a finer granular level other than a page level. Later work [19] have focused on the inefficacy of Same Origin Policy principle (SOP) implementation and enforcement in Cordova library and proposed a change on both native and JS side of the library to extend origin based access control to local resources outside the web browser. This work resulted in registering several Common Vulnerability Exploits (CVEs) [10], accordingly the library implementation has changed to overcome these limitations. Our approach distinction from other research is addressing security issues at early stage of the app development lifecycle. The proposed change can be offered as a patch or new version to the developers while they are developing their apps. The change imposes minimal effort on the developer side by automating the generation of the app configuration, which developer can use instead of the default content of the configuration file.

Hybrid Apps Specific Attacks and Solutions: Despite the newness of the approach, several papers discussed code injection attack vectors specific to Cordova-based apps and proposed solutions accordingly [24][9][25]. In their work Jin et al [24][25] have explained how XSS attacks can be mounted on mobile devices using sensors such as Contact, SMS, Barcode, MP3. These can be abused to serve as channels for receiving and spreading malicious code. The attack is based on the ability to mix code and data on web technology which can be used to inject malicious code to exploit plugin calls. To handle these attacks, they have modified Cordova plugin manager to include code that sanitizes JS calls from malicious strings. A followup work by Chen et al [9] discovered another attack channel that is based on the ability to inject malicious JS code through HTML text input fields. They have also developed a tool named DroidCIA that parses HTML files along with the JS files and sanitize malicious API calls. Our approach can prevent and detect injection attacks based on observed app behavior. It can detect dynamic injection attacks especially those include obfuscated code which could be overlooked by conventional solutions such as JS sanitization and static JS analysis.

This direction of research, where attacks are discovered and a solution is suggested to counter the mentioned attack, is still in its infancy in regard to hybrid apps. There has been a recent effort [21] to systematize analyzing potential attacks and vulnerabilities in hybrid apps by investigating common security concerns based on standardized references such as OWASP and simulate them on hybrid apps. Yet, there is still an immense need to protect the app against zero-day attacks by enforcing specific rules that conform with app healthy behavior given the expected uncertainty of the attacks channels and the how-to.

6 DISCUSSION & CONCLUSION

We presented a novel approach for configuring Cordova-based apps that considerably improves hybrid apps' security and privacy by including more aligned and fine-grained configurations that conforms with the apps' behavior. The approach help identify and fix weaknesses in a hybrid app during development to reduce attack surface. Our design adds minimal effort on developers, and still generates policies to control app behavior at run time. The policies aim to control plugin access at the state level and also controls apps behavior in terms of state transitions. Most of the implementation of this approach is done on the native side of the library which is more stable and centralized. We developed a working prototype of our modified library and used it for configuring representative applications to demonstrate viability of the change and its applicability to real-world scenarios. We have also measured overhead time of the proposed change compared to the stock version of the library, and results showed that the added overhead is minimal.

False positives are very unlikely to occur using this approach. However, false negatives are more likely to happen as the case of any approach that depends on dynamic analysis. While maintaining code coverage using an extensive test scenarios may be useful, still it is not guaranteed that it will be followed. Hence, future work should combine this approach with static analysis to achieve better automated coverage of API calls and state transitions. Nonetheless, using the proposed instrumented library still allow developers to manually modify the generated configurations the same way the stock version expects developers to do. The generated configurations can serve as the *default* configurations instead of the ones provided by the library. Developer can "tweak" it in case there is any missing configuration.

We also realize that this approach may trigger the argument of

securing by controlling the app on the expense of limiting the flexibility of the app behavior; given that dynamic code loading and updates are common practices for hybrid apps, which might question the practicality of this approach. Our counter argument is that the "security vs flexibility" dilemma is a well-known debatable subject. It is a judgment call that developers have to take. Using loose configurations as in including all plugins by default or allowing *any* URL to access and downlad resources, may be convenient and flexible but at the same time it is certainly not in favor of the device security nor users privacy.

It is important to understand that protecting apps through configuration is a vital need, but configurations alone does not guarantee full protection against attacks. Developers need more support tools that help them follow secure coding and configuration practices. Future work should extend this tool to scan Javascript and HTML code while the app is still in development and testing stage. The tool should highlight *risky* code and ultimately suggest secure coding signatures that include data sanitization and input validation especially for any data passed to device APIs through Cordova plugins. This should not only help securing the apps, it also should increase developers awareness in regard to adopting secure coding practices.

REFERENCES

[1] Apache cordova. https://cordova.apache.org.

[2] Upgrading android. https://cordova.apache.org/docs/en/4.0.0/guide/platforms/android/upgrade.html.

[3] Whitelist documentation. https://github.com/apache/cordova-plugin-whitelist.

[4] Suyesh Amatya and Arianit Kurti. Cross-platform mobile development: Challenges and opportunities. In *ICT Innovations 2013*.

[5] AppBrain. Android statistics /cordova. http://www.appbrain.com/stats/libraries/details/phonegap/phonegap-apache-cordova.

[6] Achim D Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer-Verlag.

[7] Achim D. Brucker and Michael Herzberg. On the static analysis of hybrid mobile apps: A report on the state of apache cordova nation. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*.

[8] OWASP (c). Owasp top 10. https://www.owasp.org/index.php/Top10#tab=Main.

[9] Yen-Lin Chen, Hahn-Ming Lee, Albert B Jeng, and Te-En Wei. Droidcia: A novel detection method of code injection attacks on html5-based mobile apps. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*.

[10] Mitre Corporation. Apache cordova security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-27153/Apache-Cordova.html.

[11] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*.

[12] Matt Cutts. Seo mistakes: sneaky javascript. https://www.mattcutts.com/blog/seo-mistakes-sneaky-javascript/.

[13] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Probabilistic contract compliance for mobile applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*.

[14] Apache Cordova Documentation. Whitelist guide. http://cordova.apache.org/docs/en/4.0.0/guide_appdev_whitelist_index.md.html.

[15] Nicola Dragoni, Fabio Massacci, Katsiaryna Naliuka, and Ida Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *European Public Key Infrastructure Workshop*.

[16] Apache Software Foundation. Apache software foundation. https://projects.apache.org/project.html?cordova.

[17] Gartner. Gartner recommends a hybrid approach for business-to-employee mobile apps. http://www.gartner.com/newsroom/id/2429815.

[18] Gartner. Gartner says by 2016, more than 50 percent of mobile apps deployed will be hybrid. http://www.gartner.com/newsroom/id/2324917.

[19] Martin Georgiev, Suman Jana, and Vitaly Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *NDSS symposium*.

[20] GoogleDevelopers. Making ajax applications crawable. https://developers.google.com/webmasters/ajax-crawling/docs/getting-started.

[21] Matthew L Hale and Seth Hanson. A testbed and process for analyzing attack vectors and vulnerabilities in hybrid mobile apps connected to restful web services. In *Services (SERVICES), 2015 IEEE World Congress on*.

[22] Behnaz Hassanshahi, Yaoqi Jia, Roland HC Yap, Prateek Saxena, and Zhenkai Liang. Web-to-application injection attacks on android: Characterization and detection. In *Computer Security–ESORICS 2015*.

[23] David Jaramillo, Viney Ugave, Robert Smart, and Sudeep Pasricha. Secure cross-platform hybrid mobile enterprise voice agent. In *Southeastcon 2014, Ieee*.

[24] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*.

[25] Xing Jin, Tongbo Luo, Derek G. Tsui, and Wenliang Du. Code injection attacks on html5-based mobile apps.

[26] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. Fine-grained access control for html5-based mobile applications in android. In *Information Security*.

[27] Xing Jin, Lusha Wang, Tongbo Luo, and Wenliang Du. Fine-grained access control for html5-based mobile applications in android. In *Proceedings of the 16th Information Security Conference (ISC)*.

[28] Jack Madden. Why html5 apps are ideal for enterprise mobility. http://searchmobilecomputing.techtarget.com/feature/Why-HTML5-apps-are-ideal-for-enterprise-mobility.

[29] Federico Maggi, William Robertson, Christopher Kruegel, and Giovanni Vigna. Protecting a moving target: Addressing web application concept drift. In *Recent Advances in Intrusion Detection*.

[30] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*.

[31] Parse. Parse. https://www.parse.com.

[32] D. Pitt. Mobile application architecture with html5 and javascript. http://www.infoq.com/articles/mobile-architecture-html5-javascript.

[33] Sanae Rosen, Zhiyun Qian, and Z Morely Mao. Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*.

[34] Mohamed Shehab and Abeer AlJarrah. Reducing attack surface on cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*.

[35] Kapil Singh. Practical context-aware permission control for hybrid mobile applications. In *Research in Attacks, Intrusions, and Defenses*.

[36] Kundan Singh and John Buford. Developing webrtc-based team apps with a cross-platform mobile framework. 2016.

[37] Gertner Susan Moore. Gartner says demand for enterprise mobile apps will outstrip available development capacity five to one. http://www.gartner.com/newsroom/id/3076817.

[38] Symantic. Web attack: Malicious javascript redirection 2. http://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=28341.

[39] W3C. Hash uris. http://www.w3.org/blog/2011/05/hash-uris/.

[40] W3C. State chart xml (scxml): State machine notation for control abstraction. https://www.w3.org/TR/scxml/.

[41] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*.

[42] Jing Xie, Heather Richter Lipford, and Bill Chu. Why do programmers make security errors? In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 161–164. IEEE, 2011.