On the Static Analysis of Hybrid Mobile Apps
A Report on the State of Apache Cordova Nation

Achim D. Brucker[1⋆] and Michael Herzberg[2]

[1] Department of Computer Science, The University of Sheffield, Sheffield, UK
a.brucker@sheffield.ac.uk
[2] SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
michael.herzberg@sap.com

Abstract. Developing mobile applications is a challenging business: developers need to support multiple platforms and, at the same time, need to cope with limited resources, as the revenue generated by an average app is rather small. This results in an increasing use of cross-platform development frameworks that allow developing an app once and offering it on multiple mobile platforms such as Android, iOS, or Windows.
Apache Cordova is a popular framework for developing multi-platform apps. Cordova combines HTML5 and JavaScript with native application code. Combining web and native technologies creates new security challenges as, e. g., an XSS attacker becomes more powerful.
In this paper, we present a novel approach for statically analysing the foreign language calls. We evaluate our approach by analysing the top Cordova apps from Google Play. Moreover, we report on the current state of the overall quality and security of Cordova apps.

Keywords: Static program analysis · Static application security testing · Android · Cordova · Hybrid mobile apps

1 Introduction

Developing mobile applications is a challenging business: developers need to support multiple platforms, but also have to cope with limited resources, as the revenue generated by an average app is rather small. In principle, there are three different approaches: 1) native apps, 2) mobile web apps, or 3) hybrid apps. *Native apps* are built using platform specific technologies (e. g., Swift for iOS or Java for Android). They have the advantage that they can use all platform specific features. *Mobile web apps* are on the other end of the spectrum: they are web apps developed using standard web technologies (i. e., HTML5 and JavaScript) and, thus, run on every device with a modern web browser. As a downside, they are only very shallowly, if at all, integrated into the mobile platform and can

⋆ Parts of this research were done while the author was a Security Testing Strategist and Research Expert at SAP SE in Germany.

J. Caballero et al. (Eds.): ESSoS 2016, LNCS 9639, pp. 1–17, 2016.

only access device features that are supported by HTML5. *Hybrid apps* combine the advantages of native and mobile apps; they allow developing most of the application using platform independent technologies, where small platform specific plugins enable the developer to access all device features that a native application can access.

Due to the increased market pressure for supporting multiple mobile platform as well as the increased demand to save development costs, more and more mobile apps are developed as hybrid apps. Thus, hybrid development frameworks such as PhoneGap (`http://phonegap.com/`), Trigger.io (`https://trigger.io/`), or Apache Cordova (`https://cordova.apache.org/`) are becoming more and more popular. This is not only true for small independent studios developing mobile apps, also large enterprise software vendors such as SAP are recommending the hybrid approach as the default development model to their developers. SAP offers its own extension of Apache Cordova, called SAP Kapsel, that is used both by SAP as well as its customers for developing mobile enterprise apps.

From a security development perspective, hybrid apps pose several challenges. We need to be aware that, e. g., a XSS attacker becomes much more powerful as he might be able to break out of the JavaScript environment and inject code that is executed in the context of the native part of the app—resulting in a much larger attack surface. The combination of web technologies and native mobile code is not yet supported by state of the art automated security testing tools in general and static application security testing (SAST) tools in particular. SAST tools are the back-bone of a holistic security testing strategy [3] and are widely used in the software industry [5, 6].

We address this problem by developing a static code analysis approach that supports hybrid mobile apps developed using Apache Cordova. In more detail, our contributions are twofold: 1) we present a novel technique providing the basis for detecting data-flows in hybrid mobile apps, and 2) we report on our lessons learned from applying our approach to a large number of top Cordova apps from Google Play.

2 Apache Cordova and Its Security Model

In this section, we briefly introduce Apache Cordova and provide a general overview of the particular security challenges of Cordova apps.

2.1 Apache Cordova Architecture and Programming Model

Cordova is a framework for developing mobile apps using HTML5 and JavaScript while still allowing full access to the device features.

Architecture. Fig. 1 shows the architecture of an Android Cordova app. The main part, i. e., the application logic and the user interface, are written in HTML5, CSS, and JavaScript. This part is executed in an extended WebView that provides, besides the HTML5 API, also a dedicated Cordova JavaScript

API. The latter allows, via the Cordova Native API, to access various Cordova Plugins. The Cordova Plugins are written in the platform's programming language (e. g., in Java for Android). Cordova ships with many default plugins; additional plugins are offered by third party providers or can implemented by the application developer.

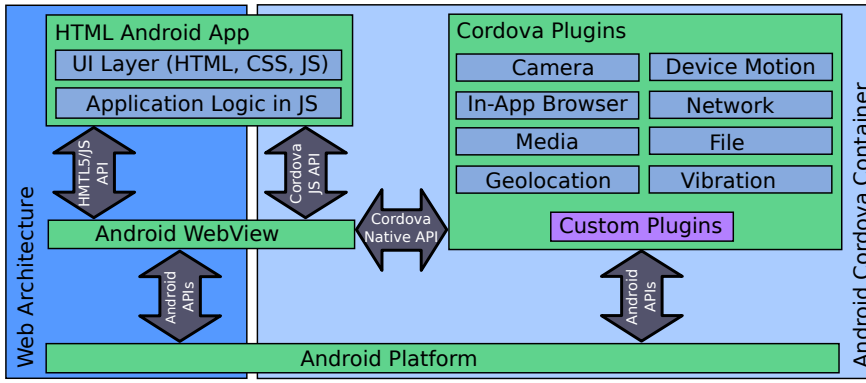

Fig. 1: The Android Cordova Architecture

Our approach also works for extensions of Cordova such as PhoneGap by Adobe or SAP Kapsel by SAP that mostly provide additional plugins.

An Example Cordova Plugin. Let us assume we want to implement a Cordova plugin that allows for searching the contacts database. Listing 1.1 (Listing 1.2) shows an excerpt of the JavaScript (Java) of the plugin implementation.

Listing 1.1 shows a JavaScript function `showPhoneNumber` that can be used to implement the business logic of a Cordova app. The `exec` method (Line 5–6) is the core of the foreign language interface of Cordova. It takes five arguments: 1. a callback that is invoked in case of a successful termination of the native call, 2. a callback that is invoked in case of a erroneous termination of the native call, 3. a string that identifies the name of Java class that implements the native

```
1 function showPhoneNumber(name) {
    var successCallback = function(contact) {
3     alert("Phone number: " + contacts.phone);
    }
5   exec(successCallback, null, "ContactsPlugin", "find",
        [{"name" : name}]);
7 }
```

Listing 1.1: Contacts Plugin Example: JavaScript

```
class ContactsPlugin extends CordovaPlugin {
  boolean execute(String action, CordovaArgs args,
                  CallbackContext callbackContext) {
    if ("find".equals(action)) {
      String name = args.get(0).name;
      find(name, callbackContext);
    } else if ("create".equals(action)) ...
  }
  void find(String name, CallbackContext callbackContext) {
    Contact contact = query("SELECT␣...␣where␣name=" + name);
    callbackContext.success(contact);
  }
}
```

Listing 1.2: Contacts Plugin Example: Java

function, 4. a string that identifies the action that should be executed by the native function, and 5. a list containing the arguments of the native function.

The Cordova framework delegates this call to the `execute` method of the Java class `ContactsPlugin` (Listing 1.2, Line 2), which delegates the call, based on the action, to the `find` method (Line 9). The `find` method uses a SQL query to find the contact information and passes it to the success callback (Line 11). The information is then passed back to the corresponding JavaScript method (Listing 1.1, Line 2).

2.2 Security Considerations for Cordova Apps

On the one hand, Cordova apps are HTML5 applications, i. e., they share all typical features (e. g., JavaScript code that is downloaded at runtime) and security risks (e. g., XSS) of web applications (see, e. g., [19, 23] for an overview of these risks). On the other hand, Cordova apps share the features (e. g., full device access) and security risk (e. g., SQL injections, privacy leaks) of native apps (see, e. g., [17, 27] for an overview of these risks).

To limit the typical web application threats, WebViews are re-using the well-known security mechanism from web browsers such as the same-origin policy [10]. Moreover, WebViews are separated from the regular web browsers on Android, e. g., WebViews have their own cache and cookie store. Still, there are subtle differences that make implementing secure Cordova apps even for experienced web application developers a challenge [9, 10].

A plugin is a mechanism for drilling holes into the sandbox of a WebView, making the traditional web attacker much more powerful as, e. g., an XSS attack might grant access to arbitrary device features. The root cause for such vulnerabilities can be located in Cordova itself (e. g., CVE-2013-4710 or CVE-2014-1882) or in programming and configuration mistakes by the app developer.

There have been several works introducing more fine-grained access control mechanism for the cross-language interface in hybrid mobile apps, particularly

Cordova, such as NoFrak [10], MobileIFC [22], and others [12, 21]. They all identified the breach of the sandbox security and that Cordova fails to restrict access to plugins by untrusted JavaScript code as the major security and privacy concern. To remedy this breach, they propose modifications to the hybrid framework which mitigate attacks by introducing fine-grained access control and modifications to Android's permission model. Apache Cordova is certainly in need of such additions. This way, existing hybrid applications could be secured without modification, reducing the potential implications of vulnerabilities such as XSS. This running time protection paired with tools helping the app developers to secure their apps in the first place, such as presented in this paper, is certainly a good combination to ensure a secure experience when using hybrid apps.

3 Static Analysis for Finding Cross-Language Flows

In this section, we present our approach for building a uniform call graph for Cordova apps with connected Java and JavaScript parts. This call graph is the basis for a cross-language data-flow analysis which enables an end-to-end static program analysis of Cordova apps.

3.1 Modelling Cordova

The usage patterns of cross-language calls depends heavily on the underlying framework, e. g., Cordova. Thus, to implement a cross-language analysis, one can either model the underlying framework or analyse the application including the cross-language framework itself. In our work, we decided to model the Cordova framework due to two reasons: 1. Modelling Cordova avoids the need for re-analysing the Cordova source code for each app and 2. data-flows within the framework code are not of interest to the app developer.

Since the official documentation regarding plugins is rather sparse, many observations are based on the officially provided plugins.

The usual cross-language control flow in a Cordova app follows a JavaScript-to-Java-to-JavaScript scheme: Starting in the JavaScript part, a call to `exec` transfers the flow to the Java side, where the requested native action is executed. When finished, the Java part calls one of the two callbacks that were passed to the `exec` call, after which the flow transfers back to the JavaScript part.

We model Cordova implicitly by four Cordova specific heuristics. The purpose of the first two is finding the JavaScript callbacks passed to the `exec` call; they are the targets of the Java-to-JavaScript call chain link. The third heuristic is concerned with finding the Java callers of this link. The fourth one filters out cross-language calls which have been reported by the first three heuristics, but are very unlikely to be correct.

The JavaScript-to-Java calls are easier to detect and thus not addressed by the heuristics, because the `exec` calls are rather static and carry enough information in their service and action parameters[3] to deduct the Java call target.

[3] For more information on the usage of these two parameters, see `https://cordova.apache.org/docs/en/latest/guide/hybrid/plugins/`

Mocking the cross-language call interface. Cordova's `exec` method is the heart of its cross-language interface, thus a precise modelling of calls to this method is key. The actual implementation of this method, `androidExec` in `cordova.js`, is not useful for detecting cross-language calls statically for at least two reasons: 1. The heavy use of dynamic language features by Cordova is challenging; e. g., the callback functions passed to `exec` are being stored by `androidExec` in a global dictionary and are only used much later. Thus, it is very hard to determine statically when the callback functions are called. As a result, these calls will not get modelled by typical building algorithms, which is fatal as they are the targets of the calls from Java-to-JavaScript. 2. The algorithms for building JavaScript call graphs are often context-insensitive. As all cross-language calls from JavaScript-to-Java are done via the one `exec` method offered by Cordova, this becomes a problem. We want to be able to relate the passed callback functions to the other parameters, which provide important information about the part on the Java side which will later call these callbacks. Therefore, context-sensitivity for the calls to `exec` is vital.

Solution. Both issues are addressed by our heuristic *ReplaceCordovaExec*, which automatically pre-processes the JavaScript source code. The core idea is to search for all `cordova.exec` and `exec` calls and replace each of them with a call to a freshly created method with a unique name that calls the callbacks.

```
function showPhoneNumber(name) {
  var succCb = function(contact) {
    alert("Number: "+contacts.phone);
  }




  exec(succCb, null,
       "ContactsPlugin",
       "find", [{"name" : name}]);
}
```

Listing 1.3: Before: Example of mocking the cross-language call interface

```
function showPhoneNumber(name) {
  var succCb = function(contact) {
    alert("Number: "+contacts.phone);
  }
  function stub1(succ, fail, service,
      action, args) {
    succ(null);
    fail(null);
  }
  stub1(succCb, null,
        "ContactsPlugin",
        "find", [{"name" : name}]);
}
```

Listing 1.4: After: Example of mocking the cross-language call interface

Recall the JavaScript part of our example, Listing 1.1. Listing 1.3 shows it again in a shorter version, and Listing 1.4 shows the modifications made by *ReplaceCordovaExec*. A new method `stub1`, which replaces the `exec` call, is introduced that makes the calls to the success and fail callbacks explicit.

The renaming of the *ReplaceCordovaExec* takes into account that the result of invoking `require("cordova/exec")` can be assigned to an arbitrary variable. This call is not shown in the example, but often used to obtain the `exec` method. Overall, *ReplaceCordovaExec* introduces local context-sensitivity into our analysis approach.

Emulating the module loading mechanism. Cordova provides its own JavaScript module mechanism, i. e., it provides two functions for structuring JavaScript code: `define` and `require`. When a Cordova app is assembled, the plugins' JavaScript code is converted into modules, and bigger plugins use those modules, too, to separate their code.

There are basically two major challenges when searching for uses of the plugins: 1. determining which object gets returned by a call to `require`, and 2. helping the call graph builder understand what is behind the global plugin variables under which Cordova makes the plugins available.

Solution. Both issues are addressed by our heuristic *ConvertModules*, which automatically pre-processes the JavaScript source code. The object that gets returned by the require call is determined by whatever gets assigned to the `module.exports` field inside the factory function. Thus, we replace the require and `module.exports` references with a global unique variable, derived from the unique module id. Now, any call graph builder will be able to track this new global object and connect the corresponding method calls. For the plugin modules, one additional transformation needs to be applied: For all global variables (there may be more than one), which are specified in the plugin's configuration file, a statement is added to the plugin definition which assigns the variable that is created by the first transformation to the queried global variable. Normally, these variables get defined at runtime when Cordova loads the plugins, but this transformation now hard-codes these definitions into the module.

```
define("com.contacts",
  function(require, exports, module){
    exports.find =
      function(succCb, name) {
        exec(succCb, null,
          "ContactsPlugin", "find",
          [{"name" : name}]);
      };
  });
...
var succCb = function(contact) {
  alert("Number:␣"+contacts.phone);
}
plugins.contacts.find(succCb,
                      "Peter");
```

Listing 1.5: Before: Example of emulating the module loading mechanism

```
define("com.contacts",
  function(require, exports, module){
    plugins.contacts.find =
      function(succCb, name) {
        exec(succCb, null,
          "ContactsPlugin", "find",
          [{"name" : name}]);
      };
  });
...
var succCb = function(contact) {
  alert("Number:␣"+contacts.phone);
}
plugins.contacts.find(succCb,
                      "Peter");
```

Listing 1.6: After: Example of emulating the module loading mechanism

Recall our phone number example: Listing 1.5 shows an exemplary definition of the contacts plug-in with an export declaration as it would look like after being imported by Cordova. We transform this export declaration using the plugin's global variable (see Listing 1.6). As a result, the relation between this global variable and the actual plugin method becomes statically apparent.

Data-flow heuristic based on action string. While the first two heuristics enable finding the *targets* of the calls from Java-to-JavaScript related to each `exec` call, finding the *callers* poses its own challenges: when execution is transferred to the Java side, the passed callback functions can be called via a `CallbackContext` Java object. This object offers three methods which get mapped to the two callback calls: `success`, `error`, and `sendPluginResult`. Given such a call somewhere on the Java side, how does one determine the possible JavaScript targets?

All `exec` calls of a plugin are mapped to a single Java `execute` method. Thus, it is not clear how calls to methods of `CallbackContext` object map to the JavaScript callbacks. During runtime, Cordova decides based on the `feature` string which class's `execute` to call, and passes the supplied `action` string to the plugin's `execute` implementation. Commonly, each `exec` call has only one possible value for each of the two parameters, so it is possible to limit the number of Java-to-JavaScript connections by utilising these context information.

Another challenge is the frequent use of the command pattern in the `execute` method, e. g., when dealing with threads. As the calls on the `CallbackContext` object are then actually done somewhere deep in the thread library, call graph builders will not attribute this call even to the `execute` method, which is a problem since the context information supplied by the JavaScript `exec` call is needed.

Solution. As `callbackContext` calls are only of interest when an `exec` call is encountered in the JavaScript code, the parameters passed to `exec` can be used as context information when looking for the `callbackContext` calls. First the Java class and its `execute` method that corresponds to the `feature` string need to be found. As Cordova keeps a mapping from those string values to Java classes, the class is looked up there.

To determine which `CallbackContext` calls are reachable from the beginning of the `execute` method, a two-fold reachability analysis is conducted for each call site. Fig. 2 illustrates the involved control flow graph and call graphs.

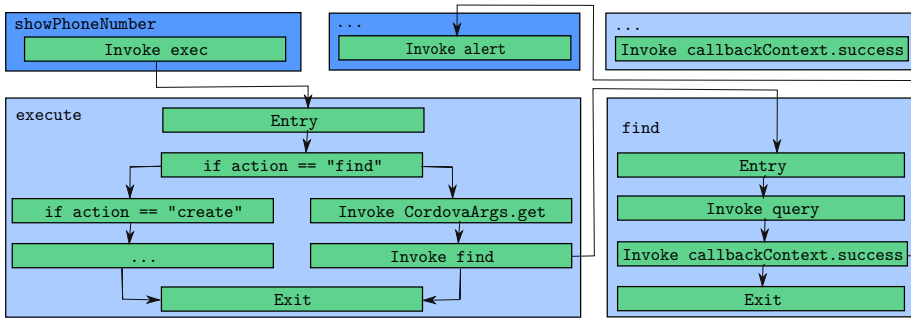

Fig. 2: The control flow graph and call graph of the example in Listing 1.1 and Listing 1.2, including two cross-language edges

1. First, using the Java call graph, we compute all possible call chains without cycles from the `execute` method to the method which contains the call to the particular `CallbackContext`. If the `execute` method is not a predecessor in this call graph, the `callbackContext` call is considered not reachable. If it is reachable though, all those invoke instructions in the `execute` method are determined through which the `callbackContext` call is eventually reachable.
2. Using these invoke instructions as well as the `action` parameter and the control flow graph of the `execute` method, a more precise reachability analysis is conducted. For each invoke instruction, all possible paths through the control flow graph from the entry of the `execute` method to the invoke are determined. For all found paths, the `action` parameter is taken into account; as many plugins implement an `execute` method using many `if-else` clauses based on `action`, the paths are checked for statements similar to `"get".equals(action)`. If the `action` strings do not match, the path can be discarded as impossible, as it can never be taken during runtime.

If there are any paths left after the two-fold reachability analysis, the reachable `callbackContext` calls need to be classified as either being a `success` or `fail` callback call. This is done by deciding whether the method called on the `CallbackContext` is either `success`, `error`, or `sendPluginResult` (and here, which status codes are possibly passed). Eventually, the corresponding `success` and `fail` connections can be reported as calls from Java-to-JavaScript.

Filtering Frameworks. The static construction of *precise* call graphs for JavaScript programs is challenging [7]. Approaches for building JavaScript call graphs have to make a compromise between scalability and correctness. Large and widely used frameworks such as jQuery (`https://jquery.com/`) or AngularJS (`https://angularjs.org`) can currently only be handled with field-based call graph builders that analyse field names non-context sensitively. Therefore, plugins that define methods with popular names or the same names as those used in the core JavaScript language such as `call`, `apply`, `get`, or `open`, result in many incorrect edges in the call graph.

Solution. The preferred solution would be to use more precise (e. g., a context-sensitive) call graph builder. Sadly, this would reduce our approach to small applications with only a few hundreds lines of JavaScript code. Alternatively, we could exclude such frameworks from our analysis. As this would make the analysis of apps based on frameworks that change the way the JavaScript code is written, e. g., frameworks promoting an asynchronous programming style, impossible, this approach is also not feasible.

Thus, we filter the problematic functions after the call graph is constructed based on further information such as the file names. This approach, on the one hand, allows balancing correctness and scalability of the static analysis. On the other hand, the configuration of the filter need to be adapted to fit new frameworks that might emerge.

3.2 Implementation

We implemented our approach, in particular a unified call graph builder for Cordova apps, using the WALA framework (`http://wala.sf.net`).[4] Our prototype allows to process Android binaries (i. e., `APK` files) directly. Using WALA's Java front-end, the analysis of Java source of Android apps can be supported easily as well. For parsing the Dalvik binary code and the JavaScript, we rely on the front-ends provided by WALA.

First, we apply the *ReplaceCordovaExec* and *ConvertModules* heuristic to the JavaScript parts of the application. Then we use WALA for building the call graphs for the JavaScript and Java parts of a Cordova app. After building the Java and JavaScript call graphs independently, we traverse both call graphs for connecting the cross-language calls. The result is a unified call graphs that allows implementing further static analysis methods that can uniformly traverse the Java and JavaScript parts of a Cordova app.

4 The State of Cordova App Security (and Quality)

In this section, we evaluate our approach for building uniform call graphs for Cordova apps as well as report on our findings based on analysing Cordova apps from the top Android app category of the Google Play Store, three Cordova apps from SAP, and one artificial app specifically written for this work. Our evaluation is two-fold in order to assess the scalability and quality of our analysis.

4.1 Popularity of Cordova and Benchmark Selection

We took the Top 1000 apps (as ranked by Google in spring 2015) from Google Play and checked if these apps contain a `config.xml` file that belongs to the Cordova framework. Using this criterion, we could identify 50 Cordova apps. Thus, according to our analysis, only 5% of the Top 1000 apps are using Cordova.

As SAP usually distributes its applications directly to its customers, we did not expect SAP apps within the Top 1000 apps category. To include SAP apps and their specific characteristics in our analysis, we have selected three mobile enterprise apps from SAP that are based on SAP Kapsel and SAP's OpenUI5 JavaScript framework (for details, see `http://openui5.org/`).

Finally, we implemented one test app, called Damn Vulnerable Hybrid Mobile App (DVHMA), that intentionally contains vulnerabilities and different coding styles to serve as a controlled test bed for our analysis.[5]

4.2 Scalability

To evaluate the runtime behaviour and, thus, the scalability of our approach, we analysed all 54 apps of our test set. Our prototype is able to analyse 52 out

[4] Our prototype is available at `https://github.com/DASPA/DASCA`.
[5] The DVHMA app is available at `https://github.com/ZertApps/DVHMA`.

of the 54: two apps from the Top 1000 are obfuscated in such a way that the WALA front-ends are not able to analyse them at all.

Our analysis can build the unified call graph for 50% of the apps in under 30 minutes and for all but one within 12 hours. The memory consumption was in all but one cases under 8 GB. The benchmarks have been run on a virtual machine running Ubuntu 14.04 using six cores of an Intel Xeon CPU E7-4830v2 @ 2.20GHz and 12 GB of RAM. Due to space reasons, we omit the detailed results. Thus, our prototype is able to analyse typical Android apps on modern modern workstations and notebooks.

In general, the runtime for building the language specific call graphs is mainly influenced by the complexity in terms of the number of function calls as call depth and only to a minor extend by the code size. This is true for both the Java as well as the JavaScript part. For building the unified call graphs, the number of cross-language calls is, given the pre-computed call graphs for each language, the main influence for building the unified call graphs.

4.3 Quality

To assess the quality of our analysis, we selected eight apps (six from the top apps as ranked by the Google Play Store, one from SAP, and our artificial test app). We did a thorough manual code review either on the original source code (for the app from SAP and our test app) or on the result of de-compiling the binary (for the six apps from Google Play). Our manual code review focused on finding all cross-language calls.

As a manual code review is a time consuming task, we limited the analysis to eight apps that we consider a good representation of the overall population of Cordova apps: Table 1 shows that the most commonly used plugins from the six apps from Google Play are the same ones as from the 50 apps. In addition, we have chosen a typical SAP app as well as our test apps that captures our expertise based on a shallow analysis of a larger number of Cordova apps. We consider the distribution of plugins as most relevant for our work, as cross-language calls are most often located in plugins. Thus, this analysis allows us to assess the quality of the unified call graphs with respect to capturing cross-language calls.

The following four sections will compare the *manually* found cross-language calls with the ones reported from the prototype. We will focus on the calls from Java-to-JavaScript. The calls from JavaScript-to-Java are relatively easy to find, thanks to the structure of Cordova's function interface. Therefore, the prototype found all these calls.

Two values are especially important when evaluating the quality [1]:

$$R = \frac{TP}{TP + FN} \quad \text{(recall)} \qquad\qquad P = \frac{TP}{TP + FP} \quad \text{(precision)}$$

where TP is the number of correctly found cross-language calls, FP the number of falsely reported ones, and FN the number of missed calls.

Table 1: The ten most used plugins from each test set

(a) Plugins from the 50 apps

Plugin	#
device	26
inappbrowser	25
dialogs	20
splashscreen	18
network-information	14
file	14
console	12
camera	11
statusbar	11
PushPlugin	11

(b) Plugins from the six manually analysed apps

Plugin	#
device	5
inappbrowser	5
dialogs	2
splashscreen	2
console	2
network-information	1
file	1
camera	1
statusbar	1
PushPlugin	1

Informally, *recall* is defined as the number of correctly found calls divided by the number of calls which should have been found and *precision* is defined as the number of correctly found calls divided by the number of calls reported.

ReplaceCordovaExec. This heuristic is necessary to identify any Java-to-JavaScript calls at all. Without it, the callback functions on the JavaScript side will not get modelled, which is bad since they are the targets of those calls from the Java side. As can be seen in Table 2a, the precision with just *ReplaceCordovaExec* is already very good. However, as is represented by the bad recall, there are also many incorrect calls being reported. But before we will present the results of *FilterJavaCallSites* and *FilterJSFrameworks*, which will lead to less errors, we will present the results for another heuristic aimed at increasing the number of found calls.

ConvertModules. The main purpose of this heuristic is to model the module mechanism and thus allow finding more calls from Java-to-JavaScript. However, this effect is only observed on one of the eight apps: our artificially created one. The explanation is simple; this heuristic enables tracking callback functions through the Cordova plugin mechanism, from the application code to the actual call to `exec`. Surprisingly, our app was the only one of those eight to create and pass callbacks from application code.

The errors for two apps are significantly reduced. This is because assigning the functions to `module.exports` is not ambiguous anymore and does not result in the field-based call graph builder vastly overestimating method invocations.

FilterJavaCallSites. Adding this heuristic, two effects can be observed in Table 2c: The number of errors is greatly reduced, but at the cost of a few

Table 2: The quality of the found cross-language calls from Java-to-JavaScript

(a) *ReplaceCordovaExec*

App	Hits	Misses	Errors	Recall	Prec.
app_{01}	4	0	400	1%	100%
app_{02}	3	0	8	28%	100%
app_{03}	30	0	5804	1%	100%
app_{04}	1	0	2315	1%	100%
app_{05}	3	0	47	6%	100%
app_{06}	246	0	1567	14%	100%
sap_{01}	3	0	32	9%	100%
DVHMA	5	5	8	39%	50%

(b) *ReplaceCordovaExec* and *ConvertModules*

App	Hits	Misses	Errors	Recall	Prec.
app_{01}	4	0	394	2%	100%
app_{02}	3	0	8	28%	100%
app_{03}	30	0	4574	1%	100%
app_{04}	1	0	1157	1%	100%
app_{05}	3	0	47	6%	100%
app_{06}	246	0	1552	14%	100%
sap_{01}	3	0	32	9%	100%
DVHMA	10	0	9	53%	100%

(c) *ReplaceCordovaExec*, *ConvertModules*, and *FilterJavaCallSites*

App	Hits	Misses	Errors	Recall	Prec.
app_{01}	3	1	397	1%	75%
app_{02}	2	1	0	100%	67%
app_{03}	28	2	2829	1%	94%
app_{04}	1	0	0	100%	100%
app_{05}	2	1	12	15%	67%
app_{06}	239	7	444	35%	98%
sap_{01}	2	1	0	100%	67%
DVHMA	10	0	0	100%	100%

(d) Using all heuristics

App	Hits	Misses	Errors	Recall	Prec.
app_{01}	3	1	6	34%	75%
app_{02}	2	1	0	100%	67%
app_{03}	28	2	2323	2%	94%
app_{04}	1	0	0	100%	100%
app_{05}	2	1	4	34%	67%
app_{06}	239	7	443	36%	98%
sap_{01}	2	1	0	100%	67%
DVHMA	10	0	0	100%	100%

cross-language calls missed. The misses come from the fact that this heuristic relies on being able to trace the `callbackContext` call back to the `execute` call. Some plugins, however, store their `CallbackContext` object for later use, e. g., when a listener for changes of the network state triggers. In these cases, other possibilities than simply discarding these call sites are also imaginable: Instead of reporting the callback functions from no `exec` call as targets, the callbacks from all `exec` calls could be reported, resulting possibly in a vast over approximation.

Most of the errors which are still reported are related to the file plugin. Here, the developers used a utility method which translates a lot of different exception types into different `callbackContext` calls. However, not all actions are able to throw all of them. This distinction is not made by this heuristic and would require a more sophisticated reachability analysis.

FilterJSFrameworks. Cordova apps contain significant amounts of framework code. As expected, this heuristic increases the recall by a great amount as can be seen in Table 2d, because cross-language calls related to these frameworks are

filtered. However, as the detection of framework code is currently only based on the file name, apps who repackage all JavaScript code into one big file will not see any improvements. Also, not all errors are related to JavaScript frameworks, so some errors coming from incorrectly found calls within the apps themselves will not get filtered.

4.4 Noticeable findings about the apps

How developers use the Cordova framework. The way the Cordova framework is used differs wildly among the 50 apps. Many apps do, in fact, use Cordova as intended: The app is written in JavaScript, the Java part is unmodified and simply loads the entry-point HTML file which is set in the Cordova configuration file. Some apps, however, significantly change the Java part. The most extreme apps do not ship any HTML or JavaScript code in the APK and simply specify one hard coded URL in Java to be loaded, which is often just the mobile version of their website, hosted in a remote location.

Some apps chose a middle ground: They may first load Activities like regular Android apps, and may embed HTML and JavaScript code only into some parts of the app, where Cordova Plugins may be used to communicate back and forth. Such irregular Cordova apps are the exception and are significantly harder to statically analyse, as they change the way Cordova is integrate into the app.

How developers use the Cordova plugins. Many plugins take callback functions and pass them through to their `exec` call. Especially for plugins which do not simply yield a result which can be passed to the success callback, e. g., when the plugin is just supposed to execute a command, there are often no fail callbacks being provided, either. Some of these actions could indeed fail, which would not get propagated through to the app code itself, though, because no fail callback has been passed. This seems to indicate a lack of proper error handling for many apps, and is one of the reasons why the *ConvertModules* heuristic did not find any additional calls in the apps.

How Cordova plugins are written. Plugins generally have the character of libraries, where the JavaScript part does rarely more than encapsulate the `exec` calls. There are also no other mechanism used to conduct cross-language calls. The official Cordova plugins adhere to these guidelines. Our work is intended for this kind of plugins.

Anyone can write Cordova plugins, and not all developers adhere to these guidelines. One found plugin, apparently written just for this specific app, does not contain any JavaScript code; instead, the `exec` calls are done right in the app code itself. Other plugins represent the other extreme and implement quite a bit of the plugin logic on the JavaScript side, which could have been as well written in Java. Again some other plugins do not even use `exec` to communicate with their Java side, but use methods which are also used internally in the Cordova framework. The reason for these unnecessary uses of workarounds remains unclear.

One plugin found in those Cordova apps is special in a different way: Combining Java and JavaScript was apparently not enough, as the APK contained some native libraries accessed via JNI to do some basic arithmetic calculations. As JSON strings get passed from the JavaScript part via Java to the C part, the attack surface gets even larger.

5 Related Work

There is a large body of work that uses static program analysis for finding security vulnerabilities in JavaScript-based web applications [11, 16, 24, 26] as well as dealing with the privacy concerns of Android apps [4, 13, 18, 20].

While cross-language calls in the form of foreign language interface such as the Java Native Interface (JNI) are not new, there are surprisingly few works that address the problem of static program analysis across such interface. Among those few there is SafeJNI [25], which statically ensures that that unsafe native code cannot bypass Java's type-safety. Another example is the work of Li and Tan [15], who developed a static analysis framework to find bugs related to the different use of exceptions in Java and native code.

The most closely related work is HybriDroid. The development of HybriDroid seems to have started by Lee et al. [14] roughly at the same time as we started our work. With HybriDroid, we share the overall goal: detecting security vulnerabilities as well as leakage of private information in hybrid mobile applications on Android. In contrast to our work, HybriDroid analyses not the cross language interface of Cordova, but the low-level interface provided by Android and does not yet support Cordova. Thus, HybriDroid works rather independently from the framework (e. g., Cordova) used for developing a hybrid app and therefore reports also cross-language calls that our approach might miss, e. g., in case a Cordova developer does use the low-level functions in addition to the mechanism offered by Cordova. In exchange, our approach allows for better explanations of found issues to Cordova developers. Moreover, we expect a better scalability of our approach. Still, as both approach are very young, it is too early for a detailed comparison of the actual implementations.

The next most closely related works are FlowDroid [2] and SCanDroid [8]. Both are tools supporting the Android life-cycle model and are able to build call graphs for *native* Android apps as well as perform a static data-flow analysis for finding security vulnerabilities as well as privacy violations. For our work, SCanDroid is of particular interest, as it is based on Wala which makes it very attractive to extend its data flow analysis to support our unified call graphs. Extending the data flow analysis of SCanDroid would require developing support for the JavaScript part of our unified call graph as well as the cross language calls. In addition, the Android life-cycle events that are specific to the JavaScript part need to be added.

6 Conclusion and Future Work

We presented a novel approach for constructing a uniform call graph for hybrid mobile apps using the Cordova framework. Our evaluations show that the generated calls graphs are, with respect to the cross language calls, very accurate. Their quality, though, depends on the used call graph builder for JavaScript.

As future work, we plan to develop a data-flow analysis (e. g., extending SCanDroid [8]) on top of the uniform call graphs that will allow for detecting programming related vulnerabilities in Cordova apps such as SQL injections and to enforce policies such as "only local JavaScript code shall be allowed to access the address book" statically, i. e., at development time.

Still, the presented approach is already applicable to real Cordova applications. When the apps from the test set have been manually examined, it quickly became apparent that any tool helping with properly programming Cordova apps is useful. One app even used a custom Cordova plugin which contains libraries written in C++ that were used by the Java code, so detecting cross-language calls does not stop at just Java and JavaScript and can certainly be extended.

Acknowledgements. We would like to thank Jens Heider and Stephan Huber from Fraunhofer SIT who provided us with the initial list of Cordova apps for our evaluation. This research was partially supported by the Federal Ministry for Education and Research (BMBF) in the context of the project ZertApps (`http://www.zertapps.de/`).

References

[1] Anderson, P.: Measuring the value of static-analysis tool deployments. Security Privacy, IEEE **10**(3), 40–47 (2012).

[2] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: PLDI '14, pp. 259–269. ACM (2014).

[3] Bachmann, R., Brucker, A.D.: Developing secure software: A holistic approach to security testing. Datenschutz und Datensicherheit (DuD) **38**(4), 257–261 (2014).

[4] Batyuk, L., Herpich, M., Camtepe, S.A., Raddatz, K., Schmidt, A.D., Albayrak, S.: Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In: Malicious and Unwanted Software (MALWARE), pp. 66–72. IEEE (2011)

[5] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**, 66–75 (2010).

[6] Brucker, A.D., Sodan, U.: Deploying static application security testing on a large scale. In: Katzenbeisser, S., Lotz, V., Weippl, E. (eds.) GI Sicherheit 2014, *Lecture Notes in Informatics*, vol. 228, pp. 91–101. GI (2014).

[7] Feldthaus, A., Schafer, M., Sridharan, M., Dolby, J., Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services. In: Software Engineering (ICSE), 2013 35th International Conference on, pp. 752–761. IEEE (2013)

[8] Fuchs, A.P., Chaudhuri, A., Foster, J.S.: SCanDroid: automated security certification of android applications. Tech. Rep. CS-TR-4991, Department of Computer Science, University of Maryland, College Park (2009)
[9] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: CSS, pp. 38–49. ACM (2012).
[10] Georgiev, M., Jana, S., Shmatikov, V.: Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In: NDSS, 2014. The Internet Society (2014).
[11] Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for AJAX intrusion detection. In: World Wide Web, pp. 561–570. ACM (2009)
[12] Jin, X., Wang, L., Luo, T., Du, W.: Fine-grained access control for HTML5-based mobile applications in Android. In: ISC. (2013)
[13] Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: Scandal: Static analyzer for detecting privacy leaks in android applications. MoST (2012)
[14] Lee, S., Dolby, J., Ryu, S.: Hybridroid: Analysis framework for Android hybrid applications (2015).
[15] Li, S., Tan, G.: Finding bugs in exceptional situations of JNI programs. In: CCS, pp. 442–452. ACM (2009)
[16] Madsen, M., Livshits, B., Fanning, M.: Practical static analysis of javascript applications in the presence of frameworks and libraries. In: Foundations of Software Engineering, pp. 499–509. ACM (2013)
[17] McGraw, G.: Software Security: Building Security In. Addison-Wesley (2006)
[18] Mohr, M., Graf, J., Hecker, M.: Jodroid: Adding android support to a static information flow control tool. In: Conference on Programming Languages (2015)
[19] Rubin, A.D., Geer Jr., D.E.: A survey of web security. Computer **31**(9), 34–41 (1998).
[20] Shabtai, A., Fledel, Y., Elovici, Y.: Automated static code analysis for classifying android applications using machine learning. In: CIS, pp. 329–333. IEEE (2010)
[21] Shehab, M., AlJarrah, A.: Reducing attack surface on Cordova-based hybrid mobile apps. In: Workshop on Mobile Development Lifecycle, pp. 1–8. ACM (2014)
[22] Singh, K.: Practical context-aware permission control for hybrid mobile applications. In: Research in Attacks, Intrusions, and Defenses, pp. 307–327. Springer (2013)
[23] Stuttard, D., Pinto, M.: The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws. John Wiley & Sons, Inc. (2011)
[24] Taly, A., Erlingsson, Ú., Mitchell, J.C., Miller, M.S., Nagra, J.: Automated analysis of security-critical JavaScript apis. In: SP, pp. 363–378. IEEE (2011)
[25] Tan, G., Appel, A.W., Chakradhar, S., Raghunathan, A., Ravi, S., Wang, D.: Safe Java native interface. In: Secure Software Engineering, pp. 97–106. (2006)
[26] Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. ACM Sigplan Notices **44**(6), 87–97 (2009)
[27] Tsipenyuk, K., Chess, B., McGraw, G.: Seven pernicious kingdoms: a taxonomy of software security errors. Security Privacy, IEEE **3**(6), 81–84 (2005).