HybriDroid: Static Analysis Framework for Android Hybrid Applications

Sungho Lee
KAIST
South Korea
eshaj@kaist.ac.kr

Julian Dolby
IBM Research
USA
dolby@us.ibm.com

Sukyoung Ryu
KAIST
South Korea
sryu.cs@kaist.ac.kr

ABSTRACT

Mobile applications (apps) have long invaded the realm of desktop apps, and hybrid apps become a promising solution for supporting multiple mobile platforms. Providing both platform-specific functionalities via native code like native apps and user interactions via JavaScript code like web apps, hybrid apps help developers build multiple apps for different platforms without much duplicated efforts. However, most hybrid apps are developed in multiple programming languages with different semantics, which may be vulnerable to programmer errors. Moreover, because untrusted JavaScript code may access device-specific features via native code, hybrid apps may be vulnerable to various security attacks. Unfortunately, no existing tools can help hybrid app developers by detecting errors or security holes.

In this paper, we present HybriDroid, a static analysis framework for Android hybrid apps. We investigate the semantics of Android hybrid apps especially for the interoperation mechanism of Android Java and JavaScript. Then, we design and implement a static analysis framework that analyzes inter-communication between Android Java and JavaScript. As example analyses supported by HybriDroid, we implement a bug detector that identifies programmer errors due to the hybrid semantics, and a taint analyzer that finds information leaks cross language boundaries. Our empirical evaluation shows that the tools are practically usable in that they found previously uncovered bugs in real-world Android hybrid apps and possible information leaks via a widely-used advertising platform.

CCS Concepts

•Theory of computation → Program analysis; •Software and its engineering → Automated static analysis;

Keywords

Android, hybrid applications, static analysis, multi-language analysis, analysis framework

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970368

1. INTRODUCTION

Mobile applications (apps) have become the major source of digital media consumption [41], but supporting multiple mobile platforms for each mobile app is a challenging task. Users are spending more time with mobile apps than desktop programs [9], major internet companies consider mobile apps "first" [43], and several vendors develop their own mobile platforms to serve mobile apps effectively such as Android by Google, iOS by Apple, and Tizen OS by Samsung and Intel. However, in order to entice users on diverse platforms, developers should develop mobile apps on the various platforms, which increases the development time and cost of mobile apps. Traditional approaches to support multiple platforms for a single app is to build either one *native app* for each platform or one *web app* for all platforms. Building native apps duplicates programming of the same app logic multiple times for different platforms but they can access platform-specific functionalities. One can build a single web app that can run on any browsers, but such a web app cannot access platform-specific information.

Taking advantage of both native apps and web apps, *hybrid apps* become a promising solution. They support user interaction via JavaScript code like web apps, and they provide device-specific features via native code just like native apps. By reusing the same user-interaction code for multiple platforms, and accessing platform-specific capabilities via native API calls, developers can build multiple apps for different platforms without much duplicated efforts. Moreover, various hybrid app development frameworks like Apache Cordova (formerly PhoneGap) [16] free developers from writing platform-specific code. OutSystems survey results in both 2014 and 2015 reported hybrid apps as the most preferred mobile app development [32, 39].

While hybrid apps simplify development efforts, they introduce additional difficulties at the same time. Because most hybrid apps are developed in multiple programming languages with different semantics, app developers should well understand such semantic differences. For example, Android hybrid apps are written in both JavaScript for user interaction and Java for Android-specific features; Java code and JavaScript code interact with each other by foreign function calls. Even though the statically typed Java and extremely dynamic JavaScript have different behaviors, the official Android documentation does not specify their interaction thoroughly [21], which may be vulnerable to programmer errors. Moreover, because untrusted JavaScript code may access device-specific features via native code, hybrid apps may be vulnerable to various security attacks [46].

In this paper, we present HybriDroid, a static analysis framework for Android hybrid apps. Among various hybrid apps on different platforms, we focus on Android hybrid apps because we can check our understanding of the Android hybrid semantics by investigating the publicly available Android source code [19]. We inspect the semantics of Android hybrid apps especially for the interoperation mechanism of Java and JavaScript. Because even the Android documentation does not fully specify the semantics, we identify the semantics via extensive testing and confirm the semantics by studying the source code. Then, we design and implement a static analysis framework that analyzes Android hybrid apps by constructing call graphs for both Java and JavaScript via an on-the-fly pointer analysis. Our implementation is built on top of WALA [27], an open-source analysis framework for Java and JavaScript. To show possible use cases of HybriDroid, we present two tools: a bug detector that identifies programmer errors due to the hybrid semantics, and a taint analyzer that finds information leaks cross language boundaries. Our empirical evaluation shows that the tools are practically usable in that they found previously uncovered bugs in real-world Android hybrid apps and possible information leaks via a widely-used advertising platform.

The contributions of this paper include the following:

- **We specify the interoperation semantics of Java and JavaScript in Android hybrid apps**. It describes the core inter-language semantics, empirically tested and confirmed by source-code investigation.
- **We present HybriDroid, an open-source static analysis framework for Android hybrid apps[1].**
- **Using HybriDroid, we develop practical tools that detect previously uncovered issues in real-world Android hybrid apps in the Google Play Store.**

In the rest of this paper, we provide a brief overview of the Android hybrid app semantics (Section 2), and describe the details of the inter-language communication mechanism (Section 3). Then, we discuss challenges in static analysis of Android hybrid apps, present how we address them in building HybriDroid (Section 4), and demonstrate two sample tools built on top of HybriDroid (Section 5). We show that the tools are practically useful in detecting real-world problems (Section 6), discuss the related work (Section 7), and conclude (Section 8).

2. OVERVIEW

We provide a high-level description of the Android hybrid app semantics (Section 2.1), how JavaScript code is evaluated from Java code (Section 2.2), and how Java and JavaScript code communicate with each other (Section 2.3).

2.1 Android Hybrid App Semantics

An Android app consists of app components that can be invoked individually, and an `Activity` is an app component denoting a single screen with a user interface. An `Activity` can show a web page to a user and support powerful web browsing features via `WebView` [21] based on the Chromium open-source project [22].

For example, Figure 1 shows a sample code that loads a web page on a `WebView`. A `WebView` instance `wv` is created on

[1]HybriDroid is available at https://github.com/wala/WALA.

```
1  public class MainActivity extends Activity {
2
3    @Override
4    protected void onCreate(
5        Bundle savedInstanceState) {
6      ...
7      WebView wv = new WebView(this);
8      wv.getSettings().setJavaScriptEnabled(true);
9      wv.setWebViewClient(new MWebViewClient());
10     wv.setWebChromeClient(new MWebChromeClient());
11     wv.loadUrl("http://www.google.com");
12     ...
13   }
14
15   class MWebViewClient extends WebViewClient {
16     @Override
17     public boolean shouldOverrideUrlLoading(
18         WebView view, String url) {
19       ...
20     }
21   }
22
23   class MWebChromeClient extends WebChromeClient {
24     @Override
25     public boolean onJsAlert(WebView view,
26         String url, String message,
27         JsResult result) {
28       ...
29     }
30   }
31
32 }
```

Figure 1: Loading a web page on `WebView`

line 7, and the next line allows the JavaScript code of `wv` to be evaluated, which is disallowed by default. On line 11, the `loadUrl` method of `wv` gets a string literal argument, which denotes a web page url, and loads the web page on `wv`. In addition to such web page urls, `loadUrl` can take addresses for local web pages denoting packed web pages in Android hybrid apps as its arguments, and load them on `WebViews`. It distinguishes the addresses for online web pages and those for local web pages by the prefixes of the addresses: if the addresses start with "`http://`" or "`https://`," they are online web pages, and if they start with "`file:///`," they are local web pages.

A `WebView` runs on its own thread to load a web page; it executes the JavaScript code in the web page asynchronously with Java code. Java code can access the status of a `WebView` by calling callback methods implemented in the `WebViewClient` and `WebChromeClient` classes. The `setWebViewClient` and `setWebChromeClient` methods on lines 9 and 10, respectively, register such instances.

The `WebViewClient` class provides callback methods, which are invoked when events that impact the rendering of the content happen [21]. For example, the `shouldOverrideUrlLoading` method (lines 16–20) is called when its first parameter `view` loads a new web page at its second parameter `url`. One can intercept url loading by overriding this method.

The `WebChromeClient` class provides callback methods, which are called when events that may impact the browser UI happen. For example, the `onJsAlert` method (lines 24–29) is called when its first parameter `view` executes the JavaScript `alert` function with the third string parameter `message` as the argument of the `alert` function on a web page at its second parameter `url`. The fourth parameter `result` provides a way for JavaScript to indicate what Java should do such as clicking a confirm or cancel button of the dialog window.

2.2 Execution of JavaScript from Java

In Android hybrid apps, Java initiates inter-operation between Java and JavaScript. Like the JavaScript `eval` function that creates code from a string value and evaluates the generated code at run time, the `WebView` class provides methods that create JavaScript code from Java string values and execute the code in the JavaScript environment.

The `WebView` class provides two such methods: `loadUrl` and `evaluateJavascript`. The `loadUrl` method not only loads a web page, but also executes JavaScript code. If its argument string value starts with "`javascript:`," the method recognizes the argument as JavaScript code and executes it in the current JavaScript environment. Similarly, the `evaluateJavascript` method creates JavaScript code from its first string argument, executes it in the current JavaScript environment, and invokes its second callback argument with the result of the JavaScript code.

2.3 Interaction between Java and JavaScript

The communication capability between the native Java code and the JavaScript code enables JavaScript to access platform-specific information. Android provides two kinds of communication between Java and JavaScript: callback communication and bridge communication.

Callback Communication.

Android hybrid app developers can use callback methods provided by `WebViewClient` and `WebChromeClient` discussed in Section 2.1 as channels for communication between Java and JavaScript. We call this kind of communication *callback communication*. Developers first choose specific events that they use for inter-communication, and override the callback methods corresponding to the selected events so that they can receive the events when they occur. Then, JavaScript code can invoke the events to invoke Java callback methods and pass some information to Java as method arguments.

For example, if a developer uses the event that loads web pages as a communication channel, the developer should implement a subclass of `WebViewClient` and override the `shouldOverrideUrlLoading` method for communication. When the JavaScript code `location.href("command")` is executed, the `shouldOverrideUrlLoading` method is invoked with the `"command"` argument as its `url` parameter, and the developer can handle `"command"` in Java. Similarly, if a developer uses the `alert` event as a communication channel, when the JavaScript code `alert("command")` is executed, the `onJsAlert` callback method can handle `"command"` in Java.

Bridge Communication.

The second mechanism called *bridge communication* supports communication between Java and JavaScript directly rather than exploiting callback methods intended for other purposes. In the bridge communication mechanism, Java code injects a Java object to the JavaScript environment of a `WebView`, and JavaScript code running on the `WebView` directly invokes methods of the Java object. More specifically, the `addJavascriptInterface` method of `WebView` injects its first argument Java object to the JavaScript environment with its second string argument as the JavaScript name of the injected Java object. We call the injected Java object an *injected object*. Then, JavaScript code can invoke methods of injected objects; The bridge communication mechanism does not allow JavaScript to access fields of injected objects.

```
1  class JSBridge {
2    @JavascriptInterface
3    public String send(String msg) {
4      return "OK";
5    }
6
7    public String getName() {
8      return "JSBridge";
9    }
10 }
11
12 @Override
13 protected onCreate(Bundle savedInstanceState) {
14   ...
15   WebView wv = new WebView(this);
16   wv.addJavascriptInterface(new JSBridge(),
17                             "bridge");
18   wv.loadUrl(
19        "file:///android_asset/www/index.html");
20   ...
21 }
```

(a) Android Java code

```
1  var result = bridge.send("example");
2  var name = bridge.getName();
```

(b) JavaScript code in `index.html`

Figure 2: Bridge communication

Figure 2 presents an example code for bridge communication. The Java code shown in Figure 2(a) declares a class named `JSBridge`, which defines two methods `send` and `getName` (lines 1–10). It injects an instance of `JSBridge` as the name `bridge` into the JavaScript environment by calling the `addJavascriptInterface` method of a `WebView` (lines 16–17). Then, the Android system creates a new JavaScript object from the injected object and attaches it to the JavaScript global object with the name `bridge` whenever a web page is loaded on the `WebView`. We call the created JavaScript object a *bridge object*. At the call site of the `loadUrl` method (lines 18–19), the `index.html` page located in the `assets/www` directory of the hybrid app is loaded on the `WebView`. The `WebView` loads the web page, makes a new JavaScript environment for the web page, creates the bridge object, attaches it to the global object of the JavaScript environment, and executes the JavaScript code in the web page. When the JavaScript code in the web page is as shown in Figure 2(b), the invocation of the method `send` of the bridge object passes the argument `"example"` to the corresponding method of the injected object, receives the result `"OK"`, and assigns it to a JavaScript variable `result`.

Note that because the `addJavascriptInterface` method enables JavaScript code to control the Java environment, it may open the gate to security vulnerabilities. Even the official Android documentation makes the following note [21]:

> Use of this method in a `WebView` containing untrusted content could allow an attacker to manipulate the host application in unintended ways, executing Java code with the permissions of the host application. Use extreme care when using this method in a `WebView` which could contain untrusted content.

Thus, since the Android version 17, JavaScript code can access only such public methods of injected objects that have the `JavascriptInterface` annotation. In Figure 2(a), the

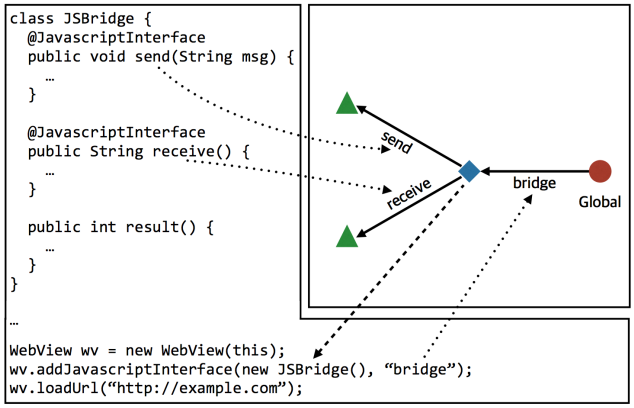

Figure 3: Bridge object initialization

method `send` has the `JavascriptInterface` annotation, but the method `getName` does not. Therefore, in Figure 2(b), the method call of `send` executes normally but the call of `getName` throws the `MethodNotFound` exception on the Android version 17 or later. Unfortunately, even recent Android should allow apps targeting old Android versions to use all public methods from JavaScript to support backward compatibility.

Because bridge communication supports interaction between Java and JavaScript more directly than callback communication, we consider only the bridge communication mechanism throughout this paper.

3. BRIDGE COMMUNICATION SEMANTICS

In this section, we describe the behavior of bridge communication in detail. Because the official Android documentation does not specify the behavior thoroughly, we identify the bridge communication semantics via extensive testing and confirm the identified semantics via source-code inspection of the Android system.

3.1 Initialization of Bridge Objects

As we discussed in Section 2.3, whenever a web page is loaded on a `WebView`, the Android system creates a bridge object for each injected object via the `addJavascriptInterface` method. It maintains a mapping between names of bridge objects and their corresponding injected objects. Thus, no multiple injected objects have the same bridge name. On the contrary, a single injected object may have multiple bridge names, which implies that multiple bridge objects may represent a single injected object.

When creating a bridge object for an injected object, the Android system generates a mock-up object for each public method annotated with `JavascriptInterface` of the injected object. We call such mock-up objects *bridge methods*. A bridge object has bridge methods as its properties named with the corresponding method names.

Figure 3 illustrates how bridge objects are initialized. The left box shows Java code and the right box shows a graph describing the JavaScript environment, where nodes denote JavaScript objects and edges denote relations between objects and names. The (red) circle is the global object in the JavaScript environment, a (blue) diamond represents a bridge object, and (green) triangles represent bridge methods. A solid edge represents that its *from* JavaScript object has its *to* JavaScript object as its property named as the edge label. A dashed edge represents that its *from* JavaScript object refers to its *to* Java object. A dotted edge represents that its *from* Java name is used in the JavaScript environment as its *to* property name. Note that the bridge object (blue diamond) refers to the injected object in Java, but the bridge methods (green triangles) do not refer to any methods in Java because Java methods are not objects. Thus, the Android system maintains only the names of bridge methods without any mapping to their corresponding Java methods.

3.2 Unique Semantics of Bridge Objects

While the Android system treats bridge objects and bridge methods similarly for normal JavaScript objects, bridge objects behave differently from normal JavaScript objects in some cases. In JavaScript, objects can dynamically add, delete, and modify their properties depending on the attributes of the properties [13]. A property associates its name with four attributes: `[[Value]]`, `[[Writable]]`, `[[Enumerable]]`, and `[[Configurable]]`. If `[[Writable]]` is true, the value of the property can change. If `[[Enumerable]]` is true, the property will be enumerated in the condition expression of the `for-in` statement. If `[[Configurable]]` is true, the property can be deleted among other things. However, even though bridge methods are properties of bridge objects and their `[[Writable]]`, `[[Enumerable]]`, and `[[Configurable]]` attributes are all true, deleting and modifying bridge methods of bridge objects are not allowed. Attempts to delete and modify bridge methods are silently ignored.

3.3 Function Calls in Bridge Communication

In Android hybrid apps, bridge method calls are different from normal JavaScript function calls. As we discussed in Section 3.1, because Java methods are not first-class objects, the Android system does not maintain mappings from bridge methods to their corresponding Java methods. Thus, when calling a bridge method, the Android hybrid mechanism finds a target Java method using Java reflection with the name of the bridge method. The evaluation of bridge method calls proceeds as follows:

1. Check whether a given receiver object is a bridge object. If it is not a bridge object, throw the `NonInjectedReceiver` exception.
2. Find a list of Java methods using Java reflection with the class name of the injected object corresponding to the receiver object and the name of the bridge method. If the list is empty, throw the `MethodNotFound` exception.
3. Find a target Java method from the method list using the number of arguments. If there is no method with the same *number of parameters (arity)*, throw the `MethodNotFound` exception. If multiple methods with the same arity exist, select a method among them.
4. Convert the JavaScript argument values to their corresponding Java values.
5. Call the target Java method with the injected object corresponding to the receiver as the `this` object and converted argument values.
6. Convert the Java method result value to its corresponding JavaScript value, and return the converted JavaScript value back to the JavaScript environment.

Table 1: Compatible types from JavaScript values to Java values

	Object	String	byte	char	short	int	long	float	double	bool	Array
Null	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
Undefined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
String	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Boolean	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
Number	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗
Object	▲*	✗	✗	✗	✗	✗	✗	✗	✗	✗	▲†

Table 2: Compatible types from Java values to JavaScript values

	Object	String	byte	char	short	int	long	float	double	bool	Array	void
JavaScript Value	▲‡	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

Note that bridge methods support method overloading *partially*. Java supports overloading which allows multiple method declarations with the same name when they have different arities or the same arity but different parameter types. However, JavaScript does not allow overloading; when there are multiple function declarations with the same name, the last one overwrites all the former ones. On the contrary, Android hybrid mechanism allows multiple bridge methods with the same name only when they have different arities as the 3rd step in the above process specifies. When multiple methods with the same name and the same arity exist, the Android system chooses a method among them as a target method to call without any warning or exception.

One interesting point is that because a bridge method maintains only the name of its corresponding Java method without any information about the class that defines the Java method, bridge method calls may produce unintuitive results. For example, consider the following JavaScript code:

```
bridge2.foo = bridge1.receive;
bridge2.foo();
```

where `bridge1` and `bridge2` are bridge objects that refer to injected objects of class `A` and class `B`, respectively; in addition, the class `A` defines a public method named `receive` with the `JavascriptInterface` annotation. Evaluation of the first line assigns the bridge method to the `foo` property of `bridge2`. Then, evaluation of the second line invokes the bridge method named `receive` with the receiver object `bridge2`, which makes the Java method named `receive` defined in the class `B` as the target Java method to call. If the class `B` happens to define a public method named `receive` with the `JavascriptInterface` annotation without any parameters, the method will be called. Otherwise, the Android system throws the `MethodNotFound` exception.

3.4 Type Mismatch in Java and JavaScript

While Java is statically typed and JavaScript is not, the evaluation process of bridge method calls involves conversion of values between Java and JavaScript. The 4th step in the bridge method call process specifies that JavaScript values denoting method arguments are converted to Java values, and the 6th step specifies that Java values denoting method call results are converted to JavaScript values.

We describe such bidirectional value conversion rules based on our extensive testing and the Android source code investigation. When a JavaScript value of JavaScript type τ_1 is passed as a method argument to a parameter of Java type τ_2, if τ_1 is compatible with τ_2 the Android system uses the value accordingly. In Table 1, when a JavaScript value of JavaScript type τ_1 such as `Null`, `Undefined`, `String`, `Boolean`, `Number`, or `Object` is passed to a parameter of Java type τ_2, the corresponding entry is marked ✓ if they are compatible, marked ▲ if the compatibility depends on the JavaScript value, and marked ✗ if they are not compatible. Similarly, Table 2 specifies whether a Java value of type τ_1 is compatible with any JavaScript value.

When JavaScript types are not compatible with Java types (✗ in Table 1), the Android system uses default values of Java types [38]; when Java types are not compatible with any JavaScript values (✗ in Table 2), the Android system uses `undefined`. The conditionally compatible cases are more complex than the others. The ▲* case is when a JavaScript object is passed as an argument to a parameter of Java object type τ_2. If the JavaScript object is not a bridge object, the Android system uses `null`. If it is a bridge object whose corresponding injected object has type τ_1, the Android system checks the type compatibility between τ_1 and τ_2. If they are compatible, it uses the injected object as the argument; otherwise, it throws an exception. When a JavaScript array value is passed as an argument to a parameter of Java object type τ_2 as in ▲†, if τ_2 is an array type with the element type τ_3, the Android system checks the compatibility of each element of the JavaScript array value and the type τ_3; otherwise, it throws an exception. Finally, when a Java object value is returned to JavaScript as in the ▲‡ case, the Android system makes and returns a new bridge object that refers to the Java object. Note that when the return type of a Java method is an array type, the Android system does not call the method but returns `undefined`, which is a peculiar semantics requiring extreme care from hybrid app developers.

4. CHALLENGES IN STATIC ANALYSIS OF ANDROID HYBRID APPS

In this section, we describe challenges in statically analyzing Android hybrid apps and how we address them. While static analysis of Android apps alone or JavaScript apps alone still has various open problems to solve, they are beyond the scope of this paper. Throughout this paper, we focus on the issues in static analysis of bridge communication between Java and JavaScript.

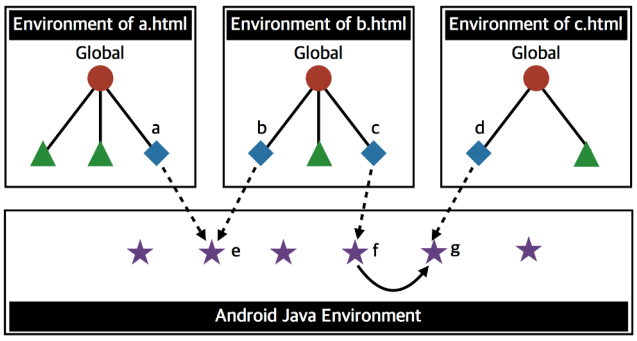

Figure 4: Modeling of JavaScript/Java environments

4.1 Multiple JavaScript Environments

Because Android hybrid apps may load multiple web pages, where each web page has its own JavaScript environment, analysis of Android hybrid apps requires analysis of interactions between a single Java environment and multiple JavaScript environments. Let us consider two possible approaches for their analysis: 1) separately analyze each web page with the Java environment, and 2) analyze all web pages with the Java environment at the same time. The first approach is simpler than the second because each analysis considers only one JavaScript environment, but it misses analysis of various hybrid behaviors. While the JavaScript environment of a web page can affect the Java environment and vice versa via bridge communication, which implies that the JavaScript environment of one web page can also affect that of another web page, the first approach cannot address such behaviors. Thus, we take the second approach.

Figure 4 illustrates our modeling of multiple JavaScript environments. To represent the JavaScript environment of each web page separately, we model that each web page has its own JavaScript global object (red circle) and the JavaScript code in one web page can access only the global object of the web page. In the Figure, (green) triangles represent normal JavaScript objects, (blue) diamonds represent bridge objects, and (purple) stars represent Java objects. Three kinds of edges represent relationships between objects: 1) solid undirected edges denote property relations, 2) dashed directed edges denote that the *from* bridge objects refer to the *to* injected objects, and 3) solid directed edges denote that the *from* object can affect the status of the *to* object directly or indirectly. Note that the JavaScript environment of each web page is isolated by its own global object. The JavaScript environments of `a.html` and `b.html` can affect each other indirectly, because their bridge objects, `a` and `b`, refer to the same injected object `e`. Also, the JavaScript environment of `b.html` can affect that of `c.html`, because an injected object `f` referred by a bridge object `c` in `b.html` can directly or indirectly affect an injected object `g` referred by `d`, a bridge object of `c.html`. On the contrary, the JavaScript environment of `c.html` can affect the Java environment, but it cannot affect any other JavaScript environments.

Our modeling soundly abstracts the Android hybrid app environments. We can further improve the modeling precision by reflecting asynchronous execution of JavaScript code in multiple `WebViews` for example. However, because our baseline analysis is flow-insensitive as we describe in Section 5, such modeling improvement belongs to our future work.

4.2 Dynamically Obtained Source Code

One of the main difficulties in static analysis of Android hybrid apps is the lack of the source code to analyze at compile time. Often, apk archives of Android hybrid apps do not contain the entire programs. While running, Android hybrid apps download and show online web pages using their addresses at run time. Also, just as in other JavaScript web apps, JavaScript code in Android hybrid apps may generate code from string values dynamically. Because such dynamically obtained code is not available statically, static analysis of the code in apk archives alone cannot analyze such code.

To analyze dynamically obtained code statically as much as possible, we perform a string analysis for Android hybrid apps prior to its main analysis. The string analysis analyzes string values used for the arguments of the `loadUrl` method, and categorizes the analysis results into four cases: 1) local web page addresses, 2) online web page addresses, 3) JavaScript code, and 4) unknown. When the argument is a local web page address, it is already available for static analysis. When the argument is an online web page address, we download the whole source code of the web page so that the main static analysis can analyze it. When the argument is JavaScript code, we include it in the analysis target of the subsequent main analysis. Finally, when the argument is not precisely known statically, we partially model it by making the main analysis analyze all local web pages in the apk archive. While this modeling is incomplete, it takes the best effort by analyzing all statically available local web pages.

The main analysis maintains mappings between the `loadUrl` call sites and their argument values to analyze each call site precisely. Even when multiple `WebViews` load the same web page, the JavaScript code in the web page may run differently depending on the `WebView` in which the web page is loaded, because each `WebView` has its own bridge objects. Therefore, for call graph construction, we identify which JavaScript environments map which bridge objects using the mapping of the `loadUrl` call sites and their argument values.

4.3 Analysis Sensitivity for Multiple Languages

The best analysis sensitivity configuration depends on several aspects. More context-sensitive analyses may produce more precise analysis results with more performance overhead for C programs [37], object-sensitive analysis may work well for Java programs [35], and more precise analysis may even improve analysis scalability for JavaScript programs [40]. Thus, it is not yet clear which analysis sensitivity would work the best for Android hybrid apps.

In order to experiment with various analysis sensitivities to better support the Android hybrid app analysis, we provide configurability of analysis sensitivities. Our analysis framework described in Section 5 supports various context-sensitive analyses to construct call graphs. It also supports different sensitivities for analyses of Java and of JavaScript.

4.4 Semantics Specific to Android Versions

Analysis of Android hybrid apps should work differently depending on the target Android versions of the apps. As we discussed in Section 2.3, in the Android version 17 or later, only public methods with the `JavascriptInterface` annotation in injected objects are accessible from JavaScript code. However, for backward compatibility, the Android system allows apps targeting old Android versions to use all public methods of injected objects from JavaScript code.

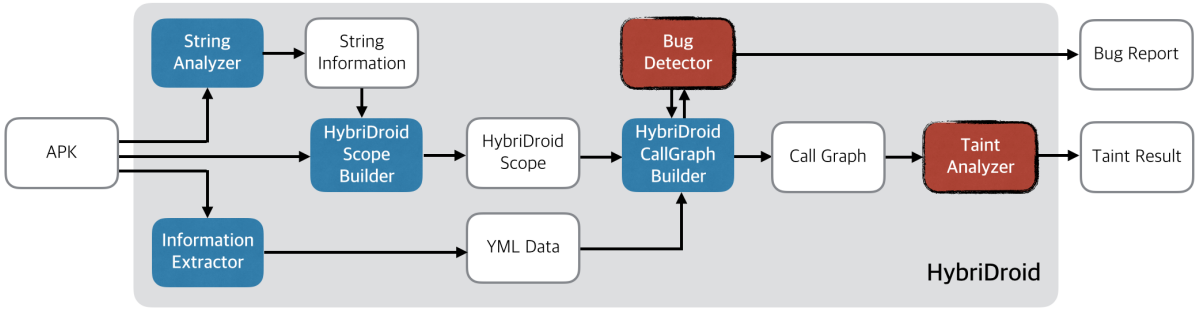

Figure 5: Overview of HybriDroid

To perform precise analysis depending on the Android versions, we extract the target version of Android hybrid apps specified in `AndroidManifest.xml` by decompiling their apk archives using `apktool` [49]. The `apktool` is a reverse engineering tool for Android apps, which decompiles Java class files to Smali [6] code, decompiles resources to their original forms, and assembles the app information into a file named `apktool.yml`. Since `apktool.yml` contains various information including the sdk, package, version, compression, and shared library, we parse `apktool.yml` to extract the target Android version of the app. Using the extracted target Android version, the analysis identifies accessible Java methods from JavaScript code for constructing call graphs.

5. HybriDroid ANALYSIS FRAMEWORK

Now, we demonstrate HybriDroid, a static analysis framework for Android hybrid apps, and show two sample tools using HybriDroid: a bug detector and a taint analyzer.

5.1 Architecture of HybriDroid

We implemented HybriDroid as an extension of WALA [27], which is an open-source static analysis framework originally developed for Java bytecode, and now it supports JavaScript programs and Android Java Dalvik bytecode. It provides flow-insensitive and context-sensitive analyses. While it can analyze JavaScript and Java separately, it cannot analyze interaction between JavaScript and Java. Thus, we use both analysis modules for JavaScript and Java, and extend them to analyze inter-communication between them.

Figure 5 presents the overall architecture of HybriDroid. For simplicity, we omit basic WALA modules and show only core modules of HybriDroid: String Analyzer, Information Extractor, HybriDroid Scope Builder, and HybriDroid CallGraph Builder. The figure also presents two add-on tools, Bug Detector and Taint Analyzer, which utilize the analysis of inter-communication between Java and JavaScript. Bug Detector finds possible programmer errors in Android hybrid apps, and Taint Analyzer tracks flows of tainted data from *sources*, origins of tainted data, to *sinks*, leaking points of tainted data. We describe the tools in detail in next sections.

String Analyzer takes the input apk archive and statically analyzes it to extract concrete string values used as the arguments of the `loadUrl` method as we discussed in Section 4.2. It first builds dependency graphs that represent how such argument values are constructed using a backward slicing technique [48]. Then, it computes string values of the `loadUrl` arguments using the dependency graph by fixpoint iterations. To focus on analysis of user-defined code, we model a set of built-in string operators such as `append` and `indexOf`.

Information Extractor retrieves necessary information of the Android hybrid apps as we described in Section 4.4. It generates YML Data, which contains various information of the target Android hybrid app in `apktool.yml`.

HybriDroid Scope Builder takes the apk and the string analysis result as inputs and builds a target analysis scope. Using the string analysis result String Information, it downloads missing web pages and saves each dynamically executed JavaScript code. It collects all the dynamically executed JavaScript code in addition to the Dalvik bytecode and JavaScript code in local web pages, and builds a HybriDroid Scope that includes all the collected code. HybriDroid Scope also contains the Android framework bytecode, Java built-in bytecode, and built-in models provided by WALA.

HybriDroid CallGraph Builder takes both HybriDroid Scope and YML Data as inputs and constructs a Call Graph for the target Android hybrid app. It extends an existing WALA module that constructs call graphs, and it handles bridge initialization, bridge communication, and multiple JavaScript environments as we discussed in Sections 3 and 4. Note that it considers the target Android version of the hybrid app from YMLData to reflect the different semantics of different Android versions. It constructs Call Graphs as the same data structure with WALA so that HybriDroid can use existing WALA analysis modules for analyzing inter-communication.

5.2 Bug Detector

Building Android hybrid apps may be more error-prone than building native apps or web apps, because no development tools like Integrated Development Environments (IDEs) exist for hybrid apps, and no documentation thoroughly specifies the delicate semantics of their inter-communication. The only possible way for developers to build high quality apps seems to be extensive testing. Moreover, because Android hybrid apps do not terminate abnormally when bugs happen in either Java or JavaScript code but only show error log messages, such bugs are difficult for developers to catch.

To help developers find such bugs, we implement Bug Detector on top of HybriDroid. When the HybriDroid CallGraph Builder constructs call graphs, Bug Detector finds possible errors and reports them. It can detect four kinds of sample bugs: MethodNotFound, TypeOverloadedBridgeMethod, NotCompatibleTypeConversion, and MethodNotExecuted.

MethodNotFound.

When a JavaScript bridge method call cannot find any target Java method to call, Bug Detector reports the MethodNotFound error, which may be due to three reasons. The first case is when a developer makes a typo in the bridge method name or indeed calls an undefined method of an injected object. The second case is when a target Java method is inaccessible either because it is private or it does not have the `JavascriptInterface` annotation when the target Android version is 17 or later. Finally, the third case is when a bridge method call has a wrong number of arguments.

MethodNotExecuted.

Bug Detector reports the MethodNotExecuted error when the return type of a target Java method is an array type. As we discussed in Section 3.4, the Android hybrid mechanism ignores such bridge method calls without any warning, which is vulnerable to programmer errors.

TypeOverloadedBridgeMethod.

Because Android hybrid apps do not support overloading with different types of parameters as we discussed in Section 3.3, Bug Detector reports the TypeOverloadedBridgeMethod warning when a bridge method call invokes a Java method that has multiple methods with the same name and number of parameters but different types. While Android does not signal an error but chooses a method among the overloaded methods, such semantics is unintuitive and error-prone.

IncompatibleTypeConversion.

Bug Detector reports the IncompatibleTypeConversion warning when a bridge method call uses default values for Java method arguments. As we discussed in Section 3.4, when the types of JavaScript values are not compatible with Java types, the default values of the corresponding Java types are used instead. While Android does not signal any error in this case, the semantics is undocumented and error-prone.

5.3 Taint Analyzer

In addition to the security vulnerabilities in Android apps and JavaScript apps, Android hybrid apps may have new kinds of security vulnerabilities involving inter-language communication. One particular example is information leakage from device features through advertising (ad) platforms [18]. Because ad platforms require various user-granted permissions and the Android system simply enumerates required permissions without actually separating execution of hybrid app code from that of ad platforms, adversaries can deploy malicious advertisement to steal private data for example.

To detect such information leaks that involve both Java and JavaScript execution flows, we implement Taint Analyzer, which tracks data and control flows in bridge communication. Specifically, it focuses on detecting information leakage via ad platforms that themselves are hybrid apps. Similarly for Bug Detector, Taint Analyzer is built on top of HybriDroid using the Interprocedural, Finite, Distributive, Subset (IFDS) framework [42] provided by WALA.

Taint Policy.

Taint Analyzer propagates taint information according to the value propagation semantics of Java, JavaScript, and bridge communication. For collection values such as arrays, lists, and maps, we over-approximate their value propagation by joining the taint information of all their elements. For example, when an array element contains a tainted value, the analysis considers the entire array as tainted. Similarly, when a native method is called with a tainted argument, the analysis considers all the arguments and its result tainted.

Sources and Sinks.

In Android hybrid apps, *sources* and *sinks* may exist cross language boundaries. For example, when an app extracts a user's location and sends it over the network via JavaScript, the source is in Java and the sink is in JavaScript. When an app receives a password from a user and saves it via Java, the source is in JavaScript and the sink is in Java.

In addition, because Android apps provide multiple layers of information flows such as inter-app flows and inter-component flows explicitly or implicitly, sources and sinks may exist cross app boundaries. For example, when an app uses an *intent* with an action `android.intent.action.PICK` to call the `Gallery` activity of the Android default app, which returns an image to the caller, the source exists in the Android default app. In this case, instead of the `Gallery` activity, we consider `android.intent.action.PICK` as the source.

Android-Specific Additional Control Flows.

In order to better analyze Android-specific flows like implicit control flows via intents and flows between sources and sinks cross app and cross component boundaries, we extend the base Android analysis of WALA with additional flows. More specifically, Taint Analyzer adds two kinds of additional edges in call graphs: 1) to support more flows between components in a single app, it adds edges from all the activity starting method calls like `startActivity` and `startActivityForResult` to all the activity creating methods `onCreate`; and 2) to support more flows between components in different apps, for each activity, it adds edges from its `startActivityForResult` calls to its `onActivityResult` to represent flows passing results from different apps' activities to the activity.

Note that because it adds extra edges on the underlying pointer analysis results, previously unreachable method calls in the pointer analysis results may become reachable due to the extra edges. Since such method calls do not have any edges to the method bodies of their corresponding callees in the pointer analysis results, the analysis of Taint Analyzer cannot analyze their method bodies. To add edges from such method calls to the bodies of their callees, we perform the Class Hierarchy Analysis (CHA) [12] from the receiver types of the calls. Traversing class extension relations from the static types of the receivers, it finds corresponding method bodies, and adds edges from the calls to the bodies.

5.4 Discussion

We presented a static analysis framework for Android hybrid apps, but there remain rooms for improvement.

String Analysis Precision.

While String Analyzer is not the main contribution of this paper, its analysis results affect the analysis quality of HybriDroid. Because we use a simple String Analyzer, HybriDroid may miss JavaScript code execution initiated by Java via `loadUrl`, and it may also miss page loading via online web pages. Note that HybriDroid is not tied to a specific string analyzer, and it can adopt advanced string analyzers [10, 31].

Table 3: Bug detection results

Rank	Hybrid App	Bug Type (#)	#FP	#TP	Bug Cause (#)	Time
1 – 100	com.gameloft.android.ANMP.GloftDMHM	MethodNotFound (1)	0	1	Obfuscation (1)	2404 sec.
	com.creativemobile.DragRacing	MethodNotFound (1)	1	0		3192 sec.
	com.gau.go.launcherex	MethodNotFound (2)	2	0		5432 sec.
	com.tripadvisor.tripadvisor	MethodNotFound (1)	0	1	Obfuscation (1)	4028 sec.
	com.dianxinos.dxbs	MethodNotFound (1)	0	1	Obfuscation (1)	1924 sec.
10,000 – 10,100	com.magmamobile.game.LostWords	MethodNotFound (1)	1	0		475 sec.
20,000 – 20,100	com.daishin	MethodNotFound (1)	0	1	Undeclared Method (1)	6572 sec.
100,000 – 100,100	com.carezone.caredroid.careapp	MethodNotFound (5)	0	5	Missing Annotation (5)	2357 sec.
	com.pateam.kanomthai	MethodNotFound (2)	0	2	Missing Annotation (2)	4209 sec.
	com.acc5.16	MethodNotFound (6)	0	6	Missing Annotation (6)	367 sec.
	jp.cleanup.android	MethodNotFound (1)	1	0		253 sec.
	ligamexicana.futbol	MethodNotFound (2)	2	0		253 sec.
200,000 – 200,100	com.sysapk.weighter	MethodNotFound (1)	0	1	Missing Annotation (1)	106 sec.
	com.youmustescape3guide.free	MethodNotFound (6)	0	6	Missing Annotation (6)	445 sec.
Total		MethodNotFound (31)	7	24	Missing Annotation (20) Obfuscation (3) Undeclared Method (1)	2287 sec.

WebView *Callback Supports.*

As we discussed in Section 2, because bridge communication supports interaction between Java and JavaScript more directly than callback communication, we consider only the bridge communication mechanism in this paper. We may want to further support callback communication.

Android Java and JavaScript Analysis.

Even though HybriDroid focuses on inter-language communication analysis, the analysis quality depends on the underlying analysis of Java and JavaScript. Because WALA supports only flow-insensitive analyses and because it does not support extensive DOM modeling, HybriDroid can further be improved by using advanced baseline analyzers.

6. EVALUATION

In this section, we show the usefulness of HybriDroid by presenting previously uncovered issues detected by Bug Detector (Section 6.1) and Taint Analyzer (Section 6.2).

6.1 Real-World Bug Detection

To evaluate the quality of Android hybrid apps in terms of the bugs defined in Section 5.2, we collected real-world Android apps using PlayDrone, a Google Play Store crawler [50]. We downloaded 100 apps each from rankings 1, 10000, 20000, 100000, and 200000, and chose hybrid apps that use bridge communication among them. We collected all 48 hybrid apps from the ranks 1 to 100, and 10 hybrid apps each for the other ranks, which amounts to 88 hybrid apps in total.

We analyzed these target hybrid apps with Bug Detector and manually verified the reported bugs as summarized in Table 3. The first column presents the ranking groups, the second column presents the apps that have reported bugs, and the remaining columns present the bug types, the numbers of unique bugs, the numbers of false positives and true positives, the causes of the bugs, and the time in seconds.

Among 88 target hybrid apps, the tool reports that 14 apps may contain 31 bugs. We observed that 9 apps contain 24 true alarms and the other 5 apps contain 7 false alarms. Surprisingly, all 24 true alarms are `MethodNotFound`. We found that hybrid app developers use bridge communication carefully without manipulating bridge objects and bridge methods; they simply call bridge methods. Moreover, most arguments to bridge methods are JavaScript strings.

Out of 24 true bugs, 20 bugs are caused by the missing `JavascriptInterface` annotation, 1 bug is because of calling an undefined method, and 3 bugs are due to wrong obfuscation. To protect Android apps from repackaging attacks [51], developers often obfuscate their apps before deployment; because obfuscation changes names of classes, methods, and fields to meaningless names, it may make reversing of the apps difficult. Google officially supports apk obfuscation by ProGuard since April 2016 [23, 26]. However, because only Java code is obfuscated, JavaScript code still accesses bridge methods using their original names even after obfuscation in Java. In order to avoid these bugs, developers should not obfuscate the accessible Java methods from JavaScript.

We observed that all 7 false positives are due to the imprecise string analysis. When the string analysis fails to find concrete values for the arguments of `loadUrl`, the tool regards that all local web pages can be loaded, which may be too conservative. We believe that a better string analysis would improve the analysis precision of HybriDroid.

6.2 Private Data Leakage Detection

To investigate security issues in ad platforms, we manually inspected all 48 hybrid apps in top 100 Android apps in the Google Play Store. We found 19 ad platforms used by them, identified 5 among them using bridge communication, and observed that 3 ad platforms (InMobi [1], Supersonic [3], and Millennial Media [34]) require rather aggressive permissions like external storage accesses and audio recording.

Among them, we closely examined InMobi, which exposes powerful Java methods including `makeCall`, `sendMail`, `takeCameraPicture`, and `getGalleryImage` [11]. When a hybrid app that integrates InMobi runs, InMobi fetches ads to the app. To analyze the InMobi ad source code, we extract the HTML and JavaScript code of the fetched ad using the Chrome remote debugging tool [20]. The extracted JavaScript `mraid.js` contains various functions that call Java methods as follows:

```
a.getGalleryImage = function() {
  return sdkController.getGalleryImage("window.imraidview")
}
```

To evaluate whether Taint Analyzer detects possible privacy leaks, we created a sample hybrid app that simply loads a

local web page, which in turn just calls the `getGalleryImage` function. We manually inspected the InMobi SDK source code and found that the above `sdkController.getGalleryImage` method call leads to a `Gallery` activity using an intent with the `android.intent.action.PICK` action. Then, the `Gallery` activity shows its image gallery. When a user selects one image, the image is sent to the JavaScript environment by calling another JavaScript function from `mraid.js`:

```
a.fireGalleryImageSelectedEvent = function(a, b, c) {
  var d = new Image;
  d.src = "" + a;
  d.width = b;
  d.height = c;
  window.imraid.broadcastEvent("galleryImageSelected", d)
}
```

via a `loadUrl` method call. We confirmed it by running the sample hybrid app with a new `fireGalleryImageSelectedEvent` function that logs its arguments.

We analyzed the sample hybrid app that uses the InMobi platform via Taint Analyzer. For a taint source, we take `android.intent.action.PICK`; for taint sinks, we consider all JavaScript functions in `mraid.js`. The tool detected the flow from `android.intent.action.PICK` to JavaScript functions and warned the possible data leakage successfully.

7. RELATED WORK

In this section, we discuss the literature on static analyses of Android apps, JavaScript programs, and hybrid apps.

Static Analysis of Android Apps.

While Android apps provide multiple layers of information flows such as inter-app flows and inter-component flows explicitly or implicitly, no existing static analyzers for Android apps can precisely analyze various flows in a scalable manner yet [47]. Most Android analyzers are built on top of either WALA or *Soot* [2]. Soot is an open-source Java and Android app analysis framework, which provides call graph construction, pointer analysis, and data-flow analysis.

SCanDroid [17] is the first WALA-based analyzer for Android apps. It tracks data flows including Inter-Component Communication (ICC) and inter-app communication to check consistency of security specification in Android manifest files. Droidel [7] provides an extensible Android analysis framework based on WALA, which generates app-specific Android framework stubs automatically.

Among various Soot-based Android analyzers, Epicc [36] provides detection of ICC vulnerabilities [14] by analyzing ICC via reducing ICC to the inter-procedural distributive environment problem. To analyze more flows in Android apps, GATOR [44] considers GUI objects, events, and callback handlers to analyze inter-component control flows induced by them. FlowDroid [5] also expands the control flows by analyzing the activity life cycle precisely. DroidSafe [24] improves precision and accuracy of Android information flow analysis by accurate analysis stubs. Apposcopy [15] proposes a new semantics-based detection of private data leakage in Android apps by specifying semantic characteristics of malware families.

Although all the above tools focus on analysis of Android apps alone, their research achievements will surely be applicable to HybriDroid regardless of their base analyzers.

Static Analysis of JavaScript Programs.

Static analysis of JavaScript programs have focused on improving the analysis scalability. JSAI [29] is a JavaScript abstract interpreter using an abstract machine-based semantics, but it supports only limited modeling of built-in, DOM, and browser APIs, which is not usable in analysis of real-world JavaScript web apps. SAFE [30] supports analysis of web apps using various JavaScript libraries, DOM and browser APIs, and platform-specific APIs and it can analyze all existing versions of jQuery. TAJS [4] tracks string values passed to the `eval` function, and it analyzes events by considering all possible combination of event calls. However, SAFE and TAJS do not support Java analysis. Finally, WALA supports both Java and JavaScript analysis. While it provides only sound pointer analysis for Java programs, it provides both sound propagation-based analysis and unsound field-based analysis for JavaScript.

Even though the implementation of HybriDroid is based on WALA, its design and the analysis between Java and JavaScript are not tied to any specific of WALA.

Static Analysis of Hybrid Apps.

Recently, static analyses of hybrid mobile apps built using cross-platform frameworks that implement the Apache Cordova library have been proposed. Jin *et al.* [28] introduced JavaScript injection attacks via network communication in hybrid mobile apps based on the PhoneGap framework, and proposed a static detection tool that analyzes only JavaScript code. Shehab and AlJarrah [45] reported security issues in the app-level access control mechanism used in Cordova-based frameworks, and developed a simple tool that checks usage of PhoneGap plugins. Brucker and Herzberg [8] built a static analyzer of hybrid mobile apps developed using Cordova. Unlike HybriDroid that analyzes flows between JavaScript and Java seamlessly, their tool builds control flow graphs of JavaScript and Java separately and adds edges between them using various heuristics.

8. CONCLUSION

Hybrid apps have become the most preferred app development because they enjoy both platform-independent user interaction via JavaScript and device-specific functionalities via native code. At the same time, hybrid app developers should be extremely careful in utilizing communication between languages with different semantics and protecting private data from malicious accesses. In this paper, we present a static analysis framework for Android hybrid apps, HybriDroid, which analyzes inter-communication between Java and JavaScript. We show that HybriDroid is useful in detecting programmer errors due to misunderstanding of inter-language communication and possible private data leakage through ad platforms. Our experimental results revealed *previously uncovered bugs* and rooms for *information leakage due to powerful functionalities of ad platforms*. We believe that HybriDroid will enable detection of various kinds of programmer errors and security vulnerabilities cross language boundaries in Android hybrid apps.

Acknowledgment

This work is supported in part by National Research Foundation of Korea (Grant NRF-2014R1A2A2A01003235) and Samsung Electronics.

9. REFERENCES

[1] InMobi. http://www.inmobi.com.

[2] Soot. https://sable.github.io/soot/.

[3] Supersonic: Mobile advertising. https://www.supersonic.com.

[4] E. Andreasen and A. Møller. Determinacy in static analysis for jQuery. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 17–31, 2014.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269, 2014.

[6] Ben Gruver. Smali. https://github.com/JesusFreke/smali.

[7] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to Android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*, pages 19–25, 2015.

[8] A. D. Brucker and M. Herzberg. On the static analysis of hybrid mobile apps. In *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems*, pages 72–88. Springer International Publishing, 2016.

[9] D. Chaffey. Mobile marketing statistics compilation. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics, 2015.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Symposium on Static Analysis*, pages 1–18, 2003.

[11] L. Constantin. Researchers: Mobile users at risk from lack of HTTPS use by mobile ad libraries. http://www.inmobi.com/company/news/researchers-mobile-users-at-risk-from-lack-of-https-use-by-mobile-ad-librar.

[12] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[13] ECMA. ECMA-262: ECMAScript 2015 Language Specification, 6th Edition. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proceedings of the 20th USENIX Conference on Security*, 2011.

[15] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587, 2014.

[16] A. S. Foundation. Apache cordova. https://cordova.apache.org, 2015.

[17] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, 2009.

[18] M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Network and Distributed System Security Symposium*, 2014.

[19] Google. Chromium. https://code.google.com/p/chromium/codesearch.

[20] Google. Remote debugging on Android with Chrome. https://developer.chrome.com/devtools/docs/remote-debugging.

[21] Google. WebView. http://developer.android.com/intl/ko/reference/android/webkit/WebView.html.

[22] Google. WebView for Android. https://developer.chrome.com/multidevice/webview/overview.

[23] Google. Shirink your code and resources. http://developer.android.com/intl/ko/tools/help/proguard.html, 2016.

[24] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *Network and Distributed System Security Symposium*, 2015.

[25] K. E. Gray. Safe cross-language inheritance. In *European Conference on Object-Oriented Programming*, pages 52–75, 2008.

[26] GuardSquare. Proguard. http://proguard.sourceforge.net.

[27] IBM. T.J. Watson libraries for analysis. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[28] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 66–77. ACM, 2014.

[29] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A static analysis platform for JavaScript. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 121–132, 2014.

[30] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages*, 2012.

[31] D. Li, Y. Lyu, M. Wan, and W. G. Halfond. String analysis for Java and Android applications. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 661–672, 2015.

[32] O. Matters. Mobile app backlog is directly damaging revenue in the enterprise. http://www.hlmtemp35.com/cm/dpl/downloads/articles/236/Mobile-Trend-Statistics-Survey-2014.pdf.

[33] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[34] M. Media. Mobile advertising and app monetization platform. http://www.millennialmedia.com.

[35] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM ToSEM*, 14(1):1–41, 2005.

[36] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. pages 543–558, 2013.

[37] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[38] Oracle. The Java™ tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html, 1995.

[39] OutSystems. State of application development report: Mobility, custom apps a priority for 2015. https://www.outsystems.com/1/mobility-custom-apps-report, 2015.

[40] C. Park and S. Ryu. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proceedings of the European Conference on Object-Oriented Programming*, 2015.

[41] S. Perez. Majority of digital media consumption now takes place in mobile apps. http://techcrunch.com/2014/08/21/majority-of-digital-media-consumption-now-takes-place-in-mobile-apps, 2014.

[42] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[43] M. Rosoff. Facebook is officially a mobile-first company. http://www.businessinsider.com/facebook-mobile-only-users-most-common-2015-11, 2015.

[44] A. Rountev and D. Yan. Static reference analysis for GUI objects in Android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[45] M. Shehab and A. AlJarrah. Reducing attack surface on Cordova-based hybrid mobile apps. In *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle*, pages 1–8. ACM, 2014.

[46] I. Software. Ensuring application security in mobile device environments. http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=WGW03009USEN&appname=WWWSEARCH.

[47] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing. Securing Android: A survey, taxonomy, and challenges. *ACM Computing Survey*, 47(4):58:1–58:45, May 2015.

[48] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[49] C. Tumbleson and R. Wiśniewski. APKTOOL. http://ibotpeaches.github.io/Apktool.

[50] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 221–233, 2014.

[51] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, pages 317–326, 2012.