BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews

Lorenzo Cavallaro
Royal Holloway, University of London

BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews

Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder
Royal Holloway, University of London
Egham, United Kingdom

***Abstract*—A Webview embeds a full-fledged browser in a mobile application and allows the application to expose a custom interface to JavaScript code. This is a popular technique to build so-called hybrid applications, but it circumvents the usual security model of the browser: any malicious JavaScript code injected into the Webview gains access to the interface and can use it to manipulate the device or exfiltrate sensitive data. In this paper, we present an approach to systematically evaluate the possible impact of code injection attacks against Webviews using static information flow analysis. Our key idea is that we can make reasoning about JavaScript semantics unnecessary by instrumenting the application with a model of possible attacker behavior—the BabelView. We evaluate our approach on 11,648 apps from various Android marketplaces, finding 2,677 vulnerabilities in 1,663 apps. Taken together, the apps reported as vulnerable have over 835 million installations worldwide. We manually validated a random sample of 66 apps and estimate that our fully automated analysis achieves a precision of 90% at a recall of 66%.**

1. Introduction

The integration of web technologies in mobile applications enables rapid cross-platform development and provides a uniform user experience across devices. Web content is usually rendered by a *Webview*, a user interface component with an embedded browser engine (`WebView` in Android, `UIWebView` in iOS). Webviews are widely used: in 2015, about 85% of applications on Google's Play Store contained one [1]. Cross-platform frameworks such as Apache Cordova, which allow to write apps entirely in HTML and JavaScript, have contributed to this high adoption and given rise to the notion of *hybrid applications*. But even otherwise native applications often embed Webviews for displaying login screens or additional web content.

Unfortunately, Webviews bring about new security threats [1], [2], [3], [4]. While the Android Webview ultimately uses WebKit [5] to render the page, the security model can be modified by app developers. Where standalone browsers enforce strong isolation, Webviews can intentionally poke holes in the browser sandbox to provide access to app- and device-specific features via a *JavaScript interface*. For instance, a hybrid banking application could provide access to account details when loading the bank's website in a Webview, or it could relay access to contacts to fill in payee details.

For assessing the overall security of an application, it is necessary to understand the implications of its JavaScript interface. When designing the interface, a developer thinks of the functionality required by her own, trusted JavaScript code executing in the Webview. However, there are several ways that an attacker can inject malicious JavaScript and access the interface [1], [6]. The observation that exposed interfaces can pose a security risk was made in previous work [7], [8]; however, some interfaces are safe and do not offer meaningful control to an attacker. In our work, we want not only to *find* Webview vulnerabilities but also to *evaluate their severity*. The intuition is that flagging up—or even removing from the marketplace—any application with an exposed JavaScript interface would be an excessive measure. If we can assess the risk posed by an application, we can focus attention on the most dangerous cases and provide meaningful feedback to developers.

We evaluate the potential impact of an attack against applications containing Webviews using a static information flow analysis that includes a model of the attacker. Our key idea is that we can avoid explicitly analyzing multiple languages by substituting the attacker's (unknown) malicious JavaScript code with a model of potential attacker behavior that over-approximates the possible information flow semantics of an attack. In particular, we instrument the target app and replace Android's Webview and its descendants with a specially crafted *BabelView* that simulates arbitrary interactions with the JavaScript interface. A subsequent information flow analysis on the instrumented application then yields new flows made possible by the attacker model, which gives an indication of the potential impact. Together with an evaluation of the difficulty of mounting an attack, this can provide an indication of the overall security risk. Our paper makes the following contributions:

- We introduce BabelView, a novel approach and implementation to evaluate the impact of code injection attacks against Webviews based on information flow analysis of applications instrumented with an attacker model.
- We analyzed 11,648 applications from the Google Play store with BabelView to evaluate our approach and report on the current state of Webview security in Android. Our analysis reports 2,677 vulnerabilities in 1,663 apps, which together are reported to have more than 835 million installations. We validate the results on a random sample of 66 applications,

and estimate the precision to be approximately 90% at a recall of 66%, confirming the practical viability of our approach.

The remainder of the paper is organized as follows. In §2, we give a brief overview of the WebView API. We then provide an overview of our approach (§3) and describe the details of our implementation (§4). We evaluate BabelView and report the results of our Android study in §5, and we discuss limitations in §6. Finally, we present related work (§7) and conclude (§8).

2. Android WebViews

We now briefly introduce the aspects of Android Webviews necessary to follow the later sections of the paper. An app can instantiate a Webview by calling its constructor or by declaring it in the Activity XML layout, from where the Android framework will create it automatically. The specifics of how the app interacts with the Webview object depend on which approach it follows; in either case, a developer can extend Android's `WebView` class to override methods and customize its behavior.

The `WebView` class offers mechanisms for interaction between the app and the web content in both directions. Java code can execute arbitrary JavaScript code in the Webview by passing a URL with the `javascript:` pseudo-protocol to the `loadUrl` method of a Webview instance. Any code passed in this way is executed in the context of the current page, just like if it were typed in a standalone browser's address bar. For the other direction and to let JavaScript code in the Webview call Java methods, the Webview allows to create custom interfaces. Any methods of an object (the *interface object*) passed to the `WebView` class's `addJavascriptInterface` method that are tagged with the `@JavascriptInterface` annotation[1] (the *interface methods*) are exported to the global JavaScript namespace in the Webview. For instance, the following example makes a single Java method available to JavaScript:

```
LocationUtils lUtils = new LocationUtils();
mWebView.addJavascriptInterface(lUtils,
    "JSlUtils");

public class LocationUtils {
  @JavascriptInterface
  public String getLocation() {
    do_something();
  }
}
```

Here, `LocationUtils` is bound to a global JavaScript object `JSlUtils` in the Webview `mWebView`. JavaScript code can access the annotated Java method `getLocation()` by calling `JSlUtils.getLocation()`.

The Webview's JavaScript interface mechanism enforces a policy of which Java methods are available to call from the JavaScript context. Developers of hybrid apps are left with a choice of what functionality to expose in an interface that is more security-critical than it appears: it is easy for a developer to misunderstand the JavaScript interface as a trusted internal interface, which is shared only between the Java and JavaScript portions of the same app; in reality it is more akin to a public API, considering the relative ease with which malicious JavaScript code can make its way into a Webview. Therefore, care must be taken to restrict the interface as much as possible and to secure the delivery of web content into the Webview. In this work we provide a way for developers and app store maintainers to detect applications with insecure interfaces susceptible to abuse; our study in §5 confirms that this is a widespread phenomenon.

3. Overview

We now provide a brief overview of our approach. We introduce the attacker model (§3.1), describe our instrumentation-based model for information flow analysis (§3.2), and discuss how the instrumentation preserves the application semantics (§3.3).

3.1. Attacker Model

Our overall goal is to identify high-impact vulnerabilities in Android applications. Our insight is that injection vulnerabilities are difficult to avoid with current mainstream web technologies, and that their presence does not justify blocking an app from being distributed to users [2]. Indeed, any standalone browser that allows loading content via insecure HTTP has this vulnerability (while some may object to calling this a "vulnerability", it clearly has security implications and has led to an increasing adoption of HTTPS by default). Following this insight, we want to pinpoint the risk of using a Webview that is embedded in an app, as opposed to a standalone browser. We assess the *degrees of freedom* an attacker gains from injecting code into a Webview with a JavaScript interface, which determine the potential impact of an injection attack.

Consequently, the attacker model for our analysis consists of arbitrary code injection into the HTML page or referenced scripts loaded in the Webview. In our evaluation, we rely, inter alia, on Fahl et al.'s work on MalloDroid [6] to estimate the difficulty of successfully mounting an injection attack (e.g., as man-in-the-middle). We note, however, that other channels are available to manipulate the code loaded in a Webview, including site-specific cross-site-scripting attacks. To abuse the JavaScript interface, the attacker then only requires the names of the interface methods, which can be obtained through reverse-engineering.

A man in the middle—who already is a strong attacker—becomes more powerful by accessing the JavaScript interface: the interface can allow to manipulate and retrieve application and device data that would not normally be part

[1] The `@JavascriptInterface` annotation was introduced in API level 17 to address a security vulnerability that allowed attackers to execute arbitrary code via the Java reflection API [9].

Algorithm 1 Information flow attacker model

```
procedure ATTACKER
    while true do
        choose iface ∈ JS-interfaces
        result ← iface(source(), source(), ...)
        sink(result)
```

of the application's network traffic. For instance, consider a hypothetical remote access application with an interface method `getProperty(key)`, which retrieves the value mapped to a key in the application's properties. That function may only ever be called with, say, the keys `"favorites"` and `"compression"`, but the attacker would be free to also use the function to retrieve the value for the key `"privateKey"`.

3.2. Instrumenting for Information Flow

Our approach is based on static information flow (or taint) analysis. We wish to find potentially dangerous information flows from injected JavaScript into security-critical parts of the Java-based app and vice-versa. At first glance, this appears to require cross-language static analysis, as recently proposed for hybrid apps [10], [11]. In this type of analysis, abstract states have to be translated between language domains, which increases complexity and can lead to a loss of precision.

However, we do not actually need to analyze JavaScript code for our purposes; rather, our attacker model assumes that the JavaScript code is under control of an attacker, so we want to model all possible actions. To this end, we propose to perform information flow analysis on the application instrumented with a representation of the attacker model in Java, such that the result is an over-approximation of all possible actions of the attacker. Our implementation replaces the Android `WebView` class (and custom subclasses) with a BabelView, a Webview that simulates an attacker specific to the app's JavaScript interfaces. We then apply a flow-, field-, and object-sensitive taint analysis [12] to detect information flows that read or write potentially sensitive information as a result of an injection attack. The BabelView provides tainted input sources to all possible sequences of interface methods and connects their return values to sinks, as shown in Algorithm 1. Here, `source()` and `sink()` are stubs that refer to sources and sinks of the underlying taint analysis.

The non-deterministic enumeration of sequences of interface method invocations is necessary since we employ a flow-sensitive taint analysis. This way, our model also covers situations where the information flow depends on a specific ordering of methods; for instance, consider the following example:

```
String id;

@JavascriptInterface
public void initialize() {
  this.id = IMEI();
}
```

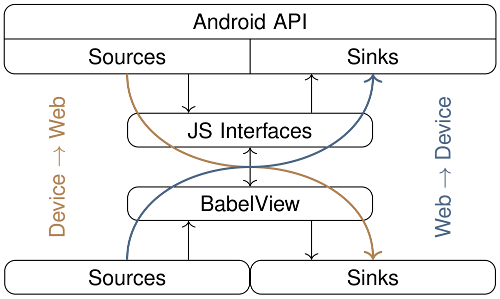

Figure 1. BabelView models flows between the attacker and sensitive sources and sinks in the Android API that cross the JavaScript interface..

```
@JavascriptInterface
public String getId() {
  return this.id;
}
```

Here, a call to `initialize` (line 4) must precede any invocation of `getId` (line 8) to cause a leak of sensitive information (the IMEI). The flow-sensitive analysis correctly distinguishes different orders of invocation, which in other places helps to reduce false positives. In the BabelView, the loop coupled with non-deterministic choice forces the analysis to join abstract states and over-approximate the result of all possible invocation orders.

Figure 1 illustrates our approach. We annotate certain methods in the Android API as sources and sinks (see §4.4), which may be accessed by methods in the JavaScript interface. The BabelView includes both a source passing data into the interface methods and a sink receiving their return values to allow detecting flows both from and to JavaScript. The source corresponds to any data injected by the attacker, and the sink to any method an attacker could use to exfiltrate information, e.g., a simple web request.

3.3. Preserving Semantics

Our instrumentation eliminates the requirement of performing a cross-language taint analysis and moves all reasoning into the Java domain. However, we need to make sure that, apart from the attacker model, the instrumentation preserves the original application's information flow semantics. In particular, we need to integrate the execution of the attacker model into the model of Android's application life cycle used as the basis of the taint analysis [12]. We solve this by overriding the methods used to load web content into the Webview (such as `loadUrl()` and `loadData()`) and modifying them to also call our attacker model (Algorithm 1). This is the earliest point at which the Webview can schedule the execution of any injected JavaScript code. The BabelView thus acts as a proxy simulating the effects of malicious JavaScript injected into loaded web content.

As the BabelView interacts only with the JavaScript interface methods, it does not affect the application's static information flow semantics in any other way than an actual JavaScript injection would. Note that this does not hold for other semantics: an application instrumented in such a way is meant only to be analyzed with static information flow analysis; in particular, the attacker model would likely crash the app if it were executed on an emulator or real device.

4. BabelView

We now describe our instrumentation-based analysis in detail. We explain the different phases of our analysis (§4.1–§4.5) and then discuss the categories of alarms that can be raised by our tool (§4.6). We implemented our static analysis and instrumentation using the Soot framework [13]; our information flow analysis relies on FlowDroid [12]. Overall, our system adds about 6 KLOC to both platforms.

4.1. Phase 1: Interface Analysis

As the first step of our analysis, we statically analyze the target application to gather information about its Webviews and JavaScript interfaces. Because we want to model an attacker accessing the interfaces of each Webview, we need to statically infer which interfaces can be added to each Webview at runtime. Our analysis is object in-sensitive: we distinguish different classes of Webviews but not different instances of the same class. For each Webview class, we determine the set of all classes of interface objects that can be added to any of the Webview class's instances.

We begin the analysis by determining all descendants of Android's `WebView` class in the application. This includes `WebView` itself, its direct or indirect subclasses, and any anonymous subclasses.

For all variables of the Webview types, we then identify calls to their `addJavascriptInterface` method. The first argument to this method is the interface object to be added, which in the method signature is of the unconstrained type `Object`. To identify the actual class used for the interface object, we determine the type T of the value passed as the first argument. We then consider as a possible class for the interface object any subtype S of T in the application that contains methods carrying the `@JavascriptInterface` annotation. To determine all interface methods accessible through S, we also have to consider supertypes of S. If any of its supertypes contain annotated methods, they are included in the set of interface objects reachable from this call to `addJavascriptInterface`. In particular, this type-based analysis handles the case where the addition of JavaScript interfaces is wrapped by framework code. Consider the following example:

```
public interface JSInterface { ... }
public class DataInterface { ... }
public class ContactInterface extends
    DataInterface implements JSInterface {
    ... }

public addInterface(JSInterface iface) {
  this.mWebView.addJavascriptInterface(iface,
      "I" + iface.getClass());
}
public void onCreate(...) {
  ...
  addInterface(new ContactInterface());
  ...
}
```

Here, our analysis would see that interface objects of type `JSInterface` can be added to the Webview and determine that the concrete type could be `ContactInterface`. It considers any annotated methods in `ContactInterface` and its ancestors, i.e., `DataInterface`. This analysis is sound but over-approximate, and can be a source of false positives.

As a result, we now have (an over-approximation of) all interface objects that can be added at each location of a call to `addJavascriptInterface`. To associate these to the correct Webview classes, we perform another type-based analysis: we associate the interface objects with all subtypes of the class of the base object on which `addJavascriptInterface` is called. This takes care of code patterns such as the following:

```
private WebView mWebView;
...
public void onCreate(...){
  ...
  mWebView = new MyWebView(this);
  mWebView.addJavascriptInterface(new
      JsObject(...))
  ...
}
```

Here, `mWebView` is of type `WebView`; the interface object `JsObject` and its ancestors will be associated with all subtypes of `WebView`, including `MyWebView`. Once again, this is a sound over-approximation but can in principle lead to false positives in applications with several custom Webview subclasses.

As the result of this phase, we now have a mapping from each descendant of the `WebView` class to a set of interface objects.

4.2. Phase 2: Generating the BabelView

We generate a `BabelView` class for each `WebView` in the mapping. Each `BabelView` becomes a subclass of its `WebView` (we remove the parent's **`final`** modifier if necessary) and overrides all of its parent's constructors so it can be used as a drop-in replacement. We make the associated interface objects explicitly available in each `BabelView`. To this end, we override the `addJavascriptInterface` method to store the interface objects passed to it in instance fields of the `BabelView` class.

To implement our attacker model, the `BabelView` needs to override all methods that load external resources and could thus be susceptible to JavaScript injection. In particular, we override `loadUrl`, `postUrl`, `loadData`, and `loadDataWithBaseURL`. The following code snippet is an example of an automatically-generated `loadUrl` method in a BabelView with a single interface object (stored in the field `jsObj`) containing two interface methods `String getPhoneID()` and `void saveFile(String, String)`.

```
@Override
public void loadUrl(String url) {
  super.loadUrl(url);
  while(true) {
    switch(random()):
    case 1:
      String id = jsObj.getPhoneID();
      leak(”getPhoneID()”, id);
    case 2:
      input = taintSource(”saveFile(...)”)
      jsObj.saveFile((String) input,
          (String) input);
  }
}
```

The method `leak` is a stub representing a tainted sink and is invoked any time a JavaScript interface returns a value. Similarly, the method `taintSource` is a stub representing a tainted input to JavaScript interfaces. Because our taint analysis is flow-sensitive, i.e., it considers the order of the statements, we need to emulate all possible combinations in which the interface methods can be called. We achieve this by generating the **`while`** loop with a non-deterministic **`switch`** statement over all interface methods (equivalent to Algorithm 1).

4.3. Phase 3: Instrumentation

In the next phase, we instrument the application to replace its Webviews with our generated BabelView instances. The instrumentation depends on how the Webview is instantiated (see §2): if it is created via an ordinary constructor call, that constructor is replaced with the corresponding constructor of its `BabelView` class. If the Webview is created via the Activity XML layout, our instrumentation searches for calls to `findViewById`, which the app has to use to obtain the Webview instance (e.g., for adding the JavaScript interface to it). To identify the calls to `findViewById` returning a Webview, our instrumenter looks for an explicit cast to a Webview class. Because we do not parse the XML layout itself, we arbitrarily choose one of the constructors of the `BabelView` (instead of the one specified in the XML). While this could potentially be a source of false positives or negatives, it would require a highly specific and unconventional design of the Webview class.

4.4. Phase 4: Information Flow Analysis

We perform a static information flow analysis on the instrumented application to identify information flows involving the attacker model. Since it relies on the FlowDroid system, the analysis is context-, flow-, field-, object-sensitive and life-cycle-aware [12]. We select sources and sinks corresponding to sensitive information sources and device functions, modified from the set provided by SuSi [14]. We further include the sources and sinks used in the BabelView classes (i.e, `taintSource` and `leak`). We show a summary of our sources and sinks in Table 1. The information flow analysis abstracts the semantics of Android framework methods. FlowDroid uses a simple modeling system (the *TaintWrapper*), where any method can either (i) be a source, (ii) be a sink, (iii) taint its object if any argument is tainted and return a tainted value if its object is tainted, (iv) clear taint from its object, (v) ignore any taint in its arguments or its object. We extended the TaintWrapper with several models that were relevant for the types of vulnerabilities we were interested in, e.g., to precisely capture the creation of Intents from tainted URIs.

Finally, we identify information flows that indicate that sensitive functionality is exposed via the JavaScript interface. For instance, a flow from `android.telephony.TelephonyManager.getDeviceId` to the method `leak` indicates that the IMEI can be retrieved from JavaScript.

4.5. Phase 5: Analysis Refinement

Preferences. The taint analysis is not precise enough to distinguish between individual key-value pairs in a map. `Preferences` are a commonly used map type in Android apps that often store sensitive information. After the information flow analysis, we refine our results by statically deriving the key values for access to preferences. Our definition of sources and sinks allows us to identify both flows from and to the `Preferences`. Given two flows f_1 and f_2, where $f_1.sink$ and $f_2.source$ are two methods to insert and retrieve values from the `Preferences`, respectively, we are interested in understanding whether (i) the type of the values is the same and (ii) the key used to access the value is the same. If these condition are met, we have an information leak from $f_1.source$ to $f_2.sink$. To determine the key values, we implemented an intra-procedural constant propagation and folding analysis for strings. In particular, we model the semantics of the `StringBuilder` class used to handle concatenation of strings. The analysis is over-approximate and may yield a top (unknown) value if it cannot determine the key to be constant.

Intents. The flow analysis cannot distinguish between different types of Intents being created; for assessing vulnerabilities, it is however important to understand the *action* of an Intent that can be controlled by an attacker. For any flow that reaches the `startActivity` sink, we perform an inter-procedural backward dependency analysis to the point of the initialization of the `Intent` until its initialization. If the Intent action is not set within the constructor, we perform a forward analysis from the constructor to find calls to `setAction` on the `Intent` object. The analysis may fail where actions are defined within intent filters (XML definitions) or through other built-in methods. To increase precision in our inter-procedural analysis, we ensure that the call-stack is consistent with an invocation through the

TABLE 1. List of sources and sinks used for the information flow analysis

Category	Sources	Sinks
TM Leaks	java.lang.String getDeviceId() java.lang.String getSubscriberId() java.lang.String getSimSerialNumber() java.lang.String getLine1Number()	void babelLeak(...)
Location Leaks	double getLatitude() double getLongitude() int getCid() int getlac()	void babelLeak(...)
Internal Connection Leaks	java.lang.String getAddress() java.lang.String getMacAddress() java.lang.String getSSID()	void babelLeak(...)
SQL-lite Leaks	java.lang.String getString(..) int getVersion() java.lang.String findEditTable(...)	void babelLeak(...)
File Opening	java.lang.Object taintSource()	java.io.File: void <init>(...)
File Reading	java.lang.String readLine()	void babelLeak(...)
File Writing	java.lang.Object taintSource()	void write(...) java.lang.Object taintSource() java.lang.Object taintSource() java.io.Writer append(...)
Intent Control	java.lang.Object taintSource()	*.Context\Activity: void startActivity[s](...)
SQL-lite Query Exec	java.lang.Object taintSource()	int delete(...) void execSQL(...) long insert(...) long insertWithOnConflict(...) android.database.Cursor .*query.*(...) long replace(...) long replaceOrThrow(...) void setForeignKeyConstraintsEnabled(...) int update(...) int updateWithOnConflict(...)
Send SMS	java.lang.Object taintSource()	void sendTextMessage(...) void sendMultiPartiTextMessage(...)
To Preferences	TM Leaks Sources SQL-lite Leaks Sources Location Leaks Sources Internal Connection Leaks Sources	*.SharedPreferences\$Editor put*(...)
From Preferences	* get*(...)	void babelLeak(...)

interface method; i.e., the interface method that triggered the flow must be reachable.

4.6. Alarm Categories

We identified five categories of potential vulnerabilities that warrant raising an alarm: information leaks, file system manipulation, Intent control, SQL-lite query execution, and sending of SMS.

Information Leaks. Flows from sources in the Android framework to a BabelView sink are categorized as leaking sensitive information. In particular, we identify different types of sensitive leaks, including telephony manager, locations, internal connections and SQL-lite.

File System Manipulation. For instance, a flow with a taint source from file data towards the BabelView sink represents a potential exfiltration of information. Similarly, a BabelView source controlling the name of a file being opened or the contents written to an open file has the potential for tampering with sensitive information.

Intent Control. Android applications use inter-component communication via Intents, often for sensitive actions. For instance, sending a phone call or a text message, accessing the contact list, opening a web page in the browser, are all achieved via Intents. If tainted input reaches a `startActivity`, then the attacker has the ability to manipulate the arguments to particular Intent actions (determined by the intent analysis in §4.5).

SQL-lite Query Execution. Applications that allow to execute SQL queries fall in this category. In particular, we look for interfaces that use external input to query an SQL-lite database.

Send SMS. This category corresponds to the ability to send text messages from the JavaScript interface via the `SmsManager`.

5. Evaluation

We now present our evaluation of BabelView and the results of our study of vulnerabilities in Android applications. Below, we explain our methodology (§5.1) and ask the following questions to evaluate our approach:

1) **Can BabelView successfully process real-world applications?** We conducted a study on a randomly selected set of applications from the AndroZoo dataset and provide a breakdown of all results, including timeouts and crashes (§5.2).
2) **Does BabelView expose real vulnerabilities?** We discuss some of the vulnerable apps in more detail to understand what an attacker can achieve under what conditions (§5.7).
3) **What are the precision and recall of our analysis?** We manually validate a random sample of apps to estimate the overall precision and recall (§5.4).

We also shed light on the current state of Webview security on Android with the following questions:

4) **How frequent are different types of vulnerabilities?** We report results both per vulnerability and per group of related vulnerabilities (§5.3).
5) **Are there types of vulnerabilities that are likely to occur in combination?** We compute the correlation between vulnerabilities and analyze our findings (§5.6).

TABLE 2. COMPOSITION BY APP STORE OF THE ANDROZOO DATASET

Market	**Number of Apps**	**Percentage**
Google	2,949,582	66.75
Anzhi	706,246	16.00
Appchina	593,125	13.42
mi.com	90,340	2.04
1Mobile	57,600	1.30
Angeeks	55,795	1.26
Slideme	52,500	1.20
Others	15,149	0.34
Total Unique	4,419,128	

5.1. Methodology

We obtained our target applications from AndroZoo [15], using the list of applications available on July 22nd, 2016, when it contained about 4.4 million applications. AndroZoo combines apps crawled from different markets, the majority of which are from the Google Play store (see Table 2 for a breakdown per store). Due to limited resources, we chose random subsets at different stages. Out of the 883,363 applications that were first seen between January 1st to July 22nd, 2016, we downloaded 209,069. Since our analysis is only meaningful on applications that contain Webviews with JavaScript interfaces, we wrote a script for Androguard[2] to select those apps that (a) ask for permission to access the Internet and (b) implement a Webview and call its `addJavascriptInterface` method. This resulted in 62,674 potential target applications (30%). From these last portion, we filter for all the applications found in Google Play Store of which we again randomly selected 11,648 applications for our evaluation.

We ran our experiments on the following:

- **S1**: A dual CPU 2.6 GHz Intel Xeon E5-2640 with 32 hyper-threads and 128 GiB of RAM
- **S2**: A single CPU 3.6 GHz i7-4790 with 8 hyper-threads and 32 GiB of RAM
- **S3**: Four 8-core CPUs 2.10 GHz Intel Xeon E5-2683 v4 with 512 GiB of RAM

[2] https://github.com/androguard/androguard

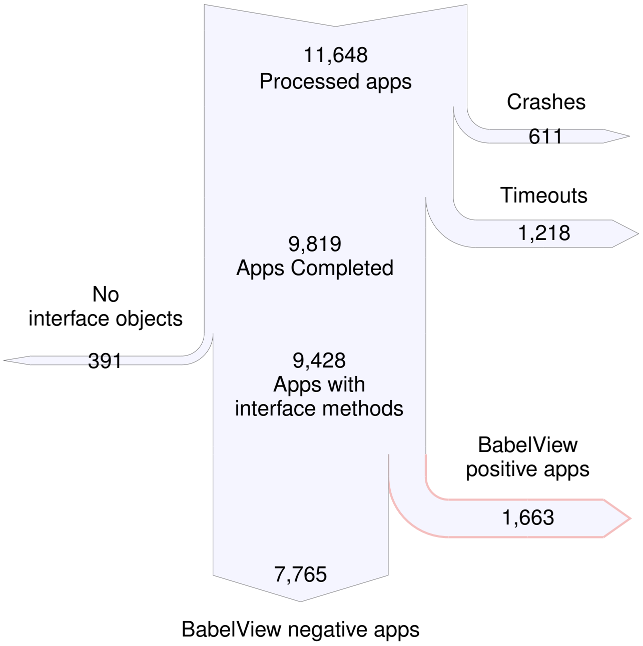

Figure 2. Breakdown of processed applications and analysis results.

- **S4**: A 32-core CPU 2.10G Hz Intel Xeon E5-2697 v3 with 256 GiB of RAM

The high precision of FlowDroid's information flow analysis can lead to long processing times on the order of hours. We set a time limit of 15 minutes for each phase of our analysis, which was a sweet spot in the sense that apps taking longer would often go on to not terminate within an hour. Each APK went through the following three phases:

1) **BabelView**: We instrumented the application with BabelView as described in §3 and §4.
2) **FlowDroid**: We ran the information flow analysis on the instrumented application.
3) **Flow Post-Processing**: We analyzed the observed flows to identify individual vulnerabilities.

Finally each APK that our analysis reported to contain potential vulnerabilities underwent the following three phases to assess the feasibility of an attack:

1) **Plain HTTP Links**: We searched the application for plain `http://` URLs.
2) **TLS Misuses**: We ran MalloDroid [6] to detect potential TLS misuse.
3) **JavaScript Injection** We stimulate each APK with the Monkey random exerciser, injecting 100 random events. We then used Bettercap to proxy the connection and perform an active MITM trying to inject JavaScript into Webviews.

5.2. Applicability

We summarize the outcome of running our tool-chain on the target dataset in Figure 2. Running BabelView on the 11,648 target applications resulted in 611 overall crashes and 1,218 taint analysis timeouts. The remaining 9,819 apps were successfully analyzed and we obtained the following results: 391 applications had no interface objects added to the declared `WebView` or no interface methods in case their target API was 17 or above; 7,765 applications had no flows involving the attacker model; and 1,663 applications were reported as positives, i.e., containing flows due to attacker behavior. Of all successfully analyzed apps implementing at least one interface object, this amounts to a rate of 17.6%.

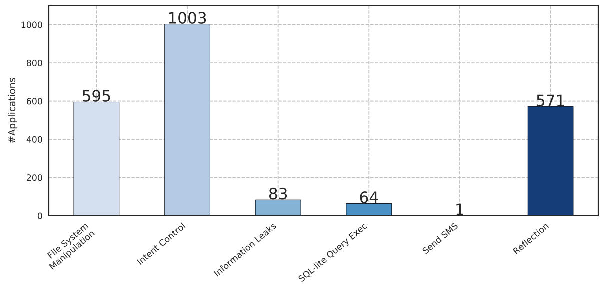

Figure 3. Number of vulnerabilities by class.

We closely analyzed all the logs of the applications that crashed in order to understand the reasons. A total of 334 applications crashed during FlowDroid's taint analysis, in all cases during the callback emulation analysis. In 89 cases the analysis failed because of missing version of the *android.jar* (i.e., android API method stubs) that Soot needs to process the APK. This affects only applications targeting API versions below 7, which are likely no longer supported by developers. The remaining 188 are generally errors due to odd byte codes or unexpected race conditions.

Among applications with interface objects, we also included those ones targeting old API versions. As shown in [16], [17], [18], using outdated version of the Android API is still common. In case of `WebViews` implementing interface objects, if an app targets API level lower than 17, it will be vulnerable to an arbitrary code execution that was reported in 2013 [9].

Applications targeting older API can still be compiled with newer versions of the SDK and therefore they can make use of the `@JavaScriptInterface` annotation (see §5.3 a more detailed explanation). Despite they are still vulnerable to the subsuming reflection vulnerability, the analysis of their interface methods is still important as these apps are likely to target API level 17 or above in future releases [4].

5.3. Vulnerabilities Found

We post-processed the analysis results for the 1,663 reported applications to identify specific vulnerabilities (see §4.4 for details on how to identify these classes). Figure 3 shows our results for high-level classes of potential vulnerabilities, i.e., information leaks, file tampering, intent control, sending SMS text messages, and SQL-lite query execution. Figure 4 shows the same results with a more detailed break

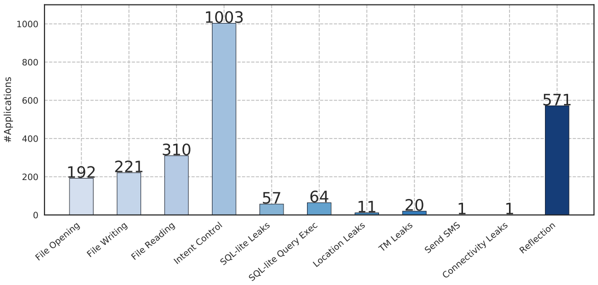

Figure 4. Number of vulnerabilities by API.

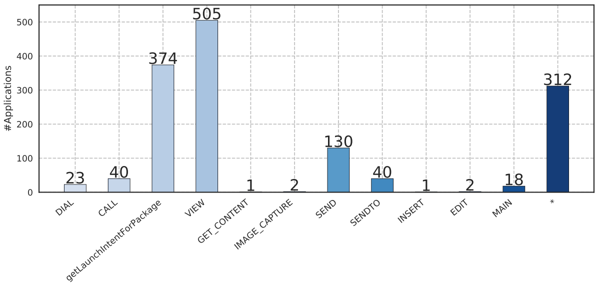

Figure 5. Types of intent control vulnerabilities. * is the global action which overapproximate the cases where our analysis could not give an answer.

down of file tampering into open, read, and write; and of information leaks into telephony manager (e.g., IMEI), location, database, and connectivity leaks. Note that the same application can have multiple vulnerabilities, which is why classes of related vulnerabilities in Figure 3 show fewer applications than the sum of individual types in Figure 4. Intuitively, file tampering vulnerabilities often go together, which we confirm in §5.6.

The results show that intent control is the most common vulnerability found: 60% of vulnerable apps allow injected JavaScript code to interact with Intents. This is followed by 36% of apps that present file system access: about 19% of apps allow reading from files, 12% writing to files and 11% opening arbitrary files. Private information leaks and the possibility of executing query on SQL-lite databases are less frequent, respectively 5% and 4%. Finally 34% of apps are still vulnerable a very dangerous reflection attack [9].

Intent control is the most common and differentiated class we identified. Indeed, depending on the nature of the controlled Intent, an attack can be more or less dangerous. Therefore, we performed a more granular analysis (see §4.5) to understand what action the controlled intent was intended to have [19]. The results of this analysis are shown in Figure 5. The action `android.action.VIEW` is most common and is found in 50% of the apps that are vulnerable to intent control. The damage an attack can do through these kind of intents is limited since the purpose of the action is to display something to the user. This is followed by the `getLaunchIntentForPackage` intent affecting 37% of the intent control apps. In this case an attack can be more severe: an attacker that can control the input for `getLaunchIntentForPackage` can start arbitrary applications on the phone. Actions `android.action.SEND` and `android.action.SENDTO`, at 13% and 4% of apps, respectively, allow an attacker to prepare the content to be sent via email or SMS. Even though an attacker can pre-fill the contents to send, further user input is required to complete the action, which lowers the potential impact. In contrast to `android.action.SENDTO`, the action `android.action.CALL` does not require any user input but can initiate a phone call immediately. This category affects 4% of apps in the intent control category.

TABLE 3. Confusion matrix, detection by app

	Label Positive	Label Negative	Total
Actual Positive	31	4	35
Actual Negative	2	29	31
Total	33	33	

TABLE 4. Confusion matrix, detection by vulnerability

Outcome	Label Positive	Label Negative	Total
Actual Positive	40	6	46
Actual Negative	10	568	578
Total	50	574	

Another common class of vulnerability is arbitrary code execution via reflection, which had been fixed since API level 17. While very problematic, this is not surprising: previous work showed that developers still implement applications targeting long superseded versions of the Android API [16], [17], [18]. Often this seems to be purely by accident. During our analysis, we observed that 410 applications of those vulnerable to reflection, despite targeting an API lower than 17, still make use of the (in this case useless) `@JavascriptInterface` notation. Android allows developers to use a newer SDK version to compile their application and still target older versions. This suggests that developers do not pay enough attention to the implications of their choice of target API level and unknowingly make their applications vulnerable to exploits that have long been avoidable.

5.4. Manual Validation

We use the manual validation to estimate the accuracy of our analysis; we are interested in two aspects:

1) Does BabelView function as an effective warning system for vulnerable hybrid apps?
2) How accurate is the BabelView attacker model with the information flow analysis in identifying individual flows?

In the first case, we sampled 33 from the 1,663 reported applications and 33 from the 7,765 negative results (see Figure 2). We manually reverse engineered (using `dex2jar` and `JD-GUI`) each application in the sample and we closely looked at each JavaScript interface method checking whether a flow due to our threat model should have been reported or not. For applications in the positive sample, we marked as true positives (TP) all those applications that were reported and for which we could verify at least one of the potential vulnerabilities—i.e., we correctly raised a warning; instead, we have a false positive (FP) if an application was reported but we verified that no potential vulnerabilities involved the app. Similarly, for a negative sample we checked if among all the non-reported potential vulnerabilities there were any that should have been detected. If yes, we marked the application as a false negative (FN). If we verified that no interface method could lead to a potential vulnerability, we marked the app as a true negative (TN). We summarize our results in the confusion matrix in Table 3.

Based on these results, we estimate the total number of TP and TN among all the reported (positive) and not reported (negative) applications. Under the assumption that the APKs are independent and equally distributed, we estimated the fraction of TP and the fraction of TN as the observed averages in our validation. In particular, we estimated that 90% of applications in our dataset will be correctly classified as positive, with a confidence interval of 8%, and that 90% will be correctly classified as negative, with a confidence interval of 11% (both at 95% confidence). We then computed the total number of TP (FP) and TN (FN) and calculated precision and recall as follows:

$$P = \frac{TP}{TP + FP} = 90\%$$

and

$$R = \frac{TP}{TP + FN} = 66\%$$

We are also interested in evaluation how accurate our analysis is for individual vulnerabilities, as opposed to flagging an entire app. We used the same positive and negative samples for this analysis as before. This time, we verified each single vulnerability, marking it as TP if it was correctly reported, as a TN if it was correctly not reported, and as FP and FN in the opposite cases. We cannot assume the independence of the vulnerabilities and do not know their distribution, so we report the results directly, without estimation, in Table 3, where the numbers now concern each type of potential vulnerability.

False Positives. Like any static analysis, BabelView can report false positives. A main source of false positives is the limited depth of tracking tainted values in fields of objects, the *access path length*. At a certain depth, taint values are merged onto the surrounding object, potentially tainting an object (and thus propagating taint further) that would not be controlled by an attacker in practice. Note further that we were conservative in our analysis of false positives: where the control flow of the (possibly obfuscated) application was too complex to determine whether there was a true flow, we assumed a false positive.

False Negatives. The main reason for having false negatives is due to the fact that the Android framework is not included in the static analysis but instead replaced by models. Therefore, during the information flow analysis some tainted values are lost. Consider for example the following snippet extracted by one of the applications we validated:

```
@JavaScriptInterface
public void openUrl(String taint){
  ...
  SSAObj sObj = new SSAObj(taint);
  String url = sObj.getString("url");
  ...
  Intent i = new Intent(Action.VIEW);
  i.setData(Uri.parse(url));
  context.startActivity(i);
}
```

`SSAObj` wraps a `JSONObject` and initializes it with `s`. However, `JSONObject` is part of the Android framework and without a model the taint will be lost unless the access path is not deep enough, and the whole `JSONObject` is tainted. In this specific case, the access path was deep enough, thus the taint never reached `startActivity`, generating a FN for intent control. This problem could in principle be solved in two different ways: (i) extending the taint wrapper for `JSONObject` so that base objects get tainted if a method listed is called with tainted parameters and (ii) have a model similar to that of `SharedPreferences` and track values in and out of a `JSONObject`. The first option would compromise precision (as now we are tainting each value in to the `JSONObject` if it is tainted). The second option comes with additional performance costs for the post-analysis, so we decided against it.

5.5. Feasibility Analysis

To better understand the feasibility of exploiting the vulnerabilities BabelView finds, we measure the difficulty of performing an injection attack. To this end, we relied on MalloDroid [6] to check applications for TLS misuse patterns, and we searched applications for hard-coded URLs beginning with the literal string `http://`, which can suggest loading web content via an insecure channel. MalloDroid reported 59.5% of vulnerable applications as using TLS insecurely. Furthermore, 97.6% of apps have hard-coded HTTP URLs, which may suggest loading HTTP content into the Webview. Although coarse-granular, such an analysis confirms the attack surface is potentially large.

To further confirm our findings, we performed a number of attacks aimed at actively injecting JavaScript code into the `WebViews`. To this end, we used Monkey to exercise the APKs, injecting 100 random events delayed of two seconds. On the injection side, we set up a MITM attack by proxying all the connections with Bettercap [20]. In order to spoof SSL traffic, we set up a certification authority on all the 4 emulators we used for the analysis. We also spoofed all the `http://` requests and tried to perform SSL stripping on the `https://` requests. The payload injected, was a print to the

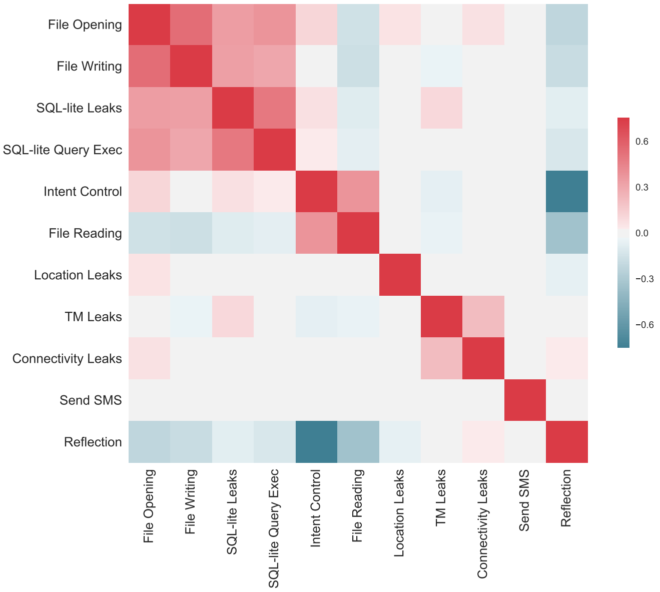

Figure 6. Correlation matrix of vulnerabilities.

emulator console. Depending on the `x-request` attribute of the request, we were able to understand whether it was made via a WebView.

The result of our analysis showed that JavaScript was executed in 360 applications over the 1,663 that BabelView reported. This result shows that injections are indeed widely possible, as suggested in previous work [6], [21].

5.6. Correlation of Vulnerabilities

We were interested to find out which vulnerabilities tend to be linked. This can highlight common patterns of functionality, but also identify interfaces where the individual vulnerabilities taken together grant additional ability to the attacker (e.g., an externally controlled file open and file write can allow to write arbitrary data to an arbitrary file).

We encoded each app as a binary vector of vulnerability types and performed a correlation analysis to find patterns of linked vulnerabilities. The resulting correlation matrix is shown as a heat map in Figure 6, with red (blue) colors corresponding to positive (negative) correlation.

As one would expect, positive correlations are found between functionally related information flows, in particular among file system and database operations. For example, file opening and writing have a correlation of 0.6, suggesting that the ability to write from arbitrary file is common. Surprisingly, file reading is not correlated with the other file system vulnerabilities. A possible explanation for this is that this is a pattern present in a common library, which we also noticed during manual validation. We also observed several cases where file reading appears together with intent control, which is also confirmed in a positive correlation of about 0.6. Other correlations exist between leaks in the telephony manager and connectivity leaks, which are semantically related. Interestingly, we also see a strong negative correlation of -0.6 between intent control and reflection vulnerabilities, and that in general the reflection vulnerability is not correlated with the other vulnerabilities. This results from the old API level not requiring an annotation on interface methods, and therefore mostly isolated detections in our analysis. Finally, send SMS and Connectivity Leaks are mostly with correlation 0 (with exception for Connectivity Leaks with TM Leaks) due to them being very uncommon.

5.7. Individual Case Studies

We now report individual case studies on a selection of apps we encountered during our analysis, to illustrate the nature of the vulnerabilities detected by BabelView. Since we are in the process of reporting the vulnerabilities to the respective developers, we have anonymized the analysis below.

Banking Application (50k installations). The JavaScript interface of this hybrid application exports several sensitive methods. The information flow analysis with BabelView flagged it as vulnerable to SQL-lite query execution, SQL-lite leaks, file writing, telephony manager leaks, and intent control. We manually reverse-engineered this application and were able to confirm all vulnerabilities. In particular, an exploit against the JavaScript interface would not only allow an attacker to place calls to arbitrary numbers and write into the file system, but also to leak messages and initiate payments. The following are some of the interface methods exposed by the application (we expand `callPhone` for illustration):

```
@JavascriptInterface
public void callPhone(String num)
{
  Intent i = new Intent(
      "android.intent.action.CALL",
      Uri.fromParts("tel", num, null));
  startActivity(i);
}

public String payFriend(...) { ... }
public String payBill(...) { ... }
public String listInbox(...) { ... }
```

We could not actively confirm the exploitability of the application in a test run, since—apart from legal reasons—the interface becomes available only after authenticating. However, from our manual analysis it is apparent that the web content displayed in the Webview is dynamically loaded.

Commercial Application (50k installations). BabelView reported this application as vulnerable to telephone manager leaks and intent control. Investigating the app, we confirmed all vulnerabilities to be true positives. In particular, we were able to invoke a method to send emails and leak the IMEI of the device through the following interface:

```
public String getDToken()
public void sendEmailWithImage(...)
```

The application implements multiple Webviews, all of them with the same JavaScript interface, and loading contents via plain HTTP. While the impact of the vulnerabilities appears less than in the case of the banking app above, the ability to send arbitrary e-mails from the victim's account can initiate identity theft or form part of social engineering.

Sports Application (500k installations). This application uses different Webviews and JavaScript interfaces; BabelView reported this application as vulnerable to SQL-lite query executions and leaks, telephone manager leaks, arbitrary file openings, and intent control. Among the different interfaces this applications used, a few were harmless and usually loaded in the application main view. The following interface methods became accessible when a user requested to create a new account:

```
public String getAccountEmail()
public String getPhoneNumber()
public String getUserPwd()
```

We successfully performed a man-in-the-middle attack on the application and injected JavaScript to access all three methods. The account e-mail and phone number are accessible immediately upon attempting to create an account. The password is stored in the phone preferences and can be retrieved through the JavaScript interface when a user visits the account creation page a second time. The underlying problem is twofold and representative for many Webview vulnerabilities: first, the Webview loads data via an insecure channel, and second, the JavaScript interface makes sensitive data available (a plaintext password). Even if the password would otherwise not be sent via the insecure channel, a JavaScript injection attack is able to retrieve it through the interface and exfiltrate it directly.

6. Limitations

Analysis. The flow analysis is subject to the same limitations as other static analysis approaches on Android. In particular, flows through native or dynamically loaded modules are typically missed. However, the nature of JavaScript interfaces and the fact that we are focusing on benign apps makes this less of an issue than when analyzing potentially malicious code.

BabelView does not distinguish WebView instances. Instead, it considers all instances of the same type to be the same object and will conservatively join the JavaScript interfaces that are added to any of the instances. We can therefore not distinguish which particular Webview instance implements a given interface method. This could hypothetically lead to false positives when the insecure interface of a Webview loading only secure resources is merged with the secure interface of an instance loading insecure resources from the web; however, we did not encounter this case in our evaluation.

Our analysis loses some sensitivity when reporting indirect leaks via `Preferences` or `Bundle`. As mentioned in §4.4, we connect sensitive flows into the application preferences with flows from the preferences to the instrumented sink method in BabelView. While this is sound and will conservatively capture any information leaks via preferences, it is not taking into account any temporal dependencies between storing and retrieving the value. A more precise treatment of this would be problematic, since preferences persist across application restarts.

Another issue with preferences is that the flow analysis effectively over-approximates the key parameter, which again can lead to false positives. A custom model to improve flow analysis via the preference map could help to improve precision further when the key parameter can be statically inferred.

Attack Feasibility. In this work, we focus on evaluating the impact of code injection attacks, but not their feasibility. We refer to the literature for a more thorough treatment of this issue [7], [22]. Here, we perform a cursory analysis of attack feasibility using MalloDroid and a lightweight search for constant HTTP URLs in the application code.

However, both are neither sufficient nor necessary for an attack to be feasible. MalloDroid's detected TLS misuse could be dead code protected by a debug flag, and HTTP URLs may be used otherwise. Furthermore, we would need to establish that the injection vulnerabilities are indeed linked to the Webview with vulnerable interfaces. In our case studies we manually confirmed the injection vulnerability, acting as a man in the middle. An automatic, dynamic validation of the attack feasibility was out of scope of this work, however. But as stated in §3, apart from man-in-the-middle attacks there are other vectors for code injection, some of which, such as cross-site-scripting, do not depend on the application.

7. Related Work

We now give an overview of some of the relevant work that has been carried out on the problem of Webview security and hybrid applications.

Webview: Attacks and Vulnerabilities. Webview vulnerabilities have been widely studied [1], [2], [3], [8], [23], [24]. Luo et al. give a detailed overview of several classes of attacks against Webviews [2], providing a basis for our work. Neugschwandtner et al. [23] were the first to highlight the magnitude of the problem. In their analysis, they categorize as vulnerable all applications implementing JavaScript interfaces and misusing TLS (or not using it at all). For further precision, they analyzed permissions and discovered that 76% of vulnerable applications requested privacy critical permissions. While this is a sign of poorly designed applications, the impact of an injection exploit very much depends on the JavaScript interfaces, motivating the work in this paper. A step forward towards this was made by Bifocals [24], a static analysis tool able to identify and evaluate vulnerabilities in Webviews. Bifocals looks for potential Webviews vulnerabilities (i.e uses JavaScript interfaces and loads third parties web pages) and then performs an impact analysis on the JavaScript interfaces. In particular, it analyzes whether these methods reach code requiring security-relevant permissions. However, JavaScript interfaces can be pose an (application-specific) risk without making use of permissions. In addition, not all JavaScript interfaces that make use of permissions are dangerous: for example, an interface method might use the phone's IMEI to perform an operation but not return it to the caller.

A large scale study on mobile web applications and their vulnerabilities was presented by Mutchler et al. [1], albeit without going into detail about the the nature of the exposed JavaScript interfaces.

The means by which malicious code can be injected into the Webview has been discussed in previous work [7], [22]. Having to interact with many forms of entities, HTML5-based hybrid applications expose a broader surface of attack, introducing new channels of injection for cross-site-scripting attacks [22]. While these attacks require the user to directly visit the malicious page within the Webview, Web- to-Application injection attacks (W2AI) relies on intent hyperlinks to render the payload simply by clinking a link in the default browser [7]. Both discuss the threat behind JavaScript interfaces, but stop their analysis at the moment where the malicious payload is loaded, without analyzing the implication of the attacker executing the JavaScript interfaces.

Static Analysis of Hybrid-Applications. As hybrid applications are becoming more common, new cross-language static analysis approaches have been developed. Brucker and Herzberg proposed a technique for building an uniform call graph for hybrid applications written against the Apache Cordova framework [10]. A more general approach was introduced in Hybridroid [11], a tool for flow analysis of hybrid applications. In this work, the authors provided a semantics for the interoperation of Java and JavaScript in Android hybrid apps without modeling a particular framework. The authors used Hybridroid for bug-finding in a potentially dangerous library using JavaScript interfaces. Static analysis approaches that take the JavaScript components into account work well if all JavaScript code is available at the time of analysis—for applications that load foreign code, it is necessary needs to construct an over-approximating model, like in our implementation of BabelView.

Webview Access Control. There have been several proposals to bring origin-based access control to Webviews [25], [26], [27]. Shehab et al. proposed a framework that modifies Cordova, enabling developers to build and enforce a page-based plugin access policy. In this way, depending on the page loaded, it will or will not have the permission to use exposed Cordova plugins (i.e., JavaScript interfaces). Georgiev et al. presented NoFrank [25], a system to extend origin-based access control to local resources outside the web browser. In particular, the application developer whitelists origins that are then allowed to access device's resources. However, once an origin is whitelisted, it can access any resource exposed. A fine-granular solution was proposed in [28]. The authors designed a system that allows developers to assign different permissions to different frames in the Webview. Tuncay et al. recently presented Draco, an even more fine-granular approach [26]. Draco defines a policy language that developers can use to design access control policies on different channels—i.e. the interface object, the event handlers and the HTML5 API. This policies will then be enforced by Draco's runtime system. While this approach can be effective in protecting the final user from loading malicious web code, the developer has the burden of designing correct policies. Moreover, defining such policy could be unnecessary if the JavaScript interfaces exposed are harmless. Therefore, we believe that such an approach could be effectively combined with BabelView to target policy enforcement at relevant interfaces.

8. Conclusion

In this paper, we presented a novel method to use information flow analysis to evaluate the possible impact of code injection attacks against mobile applications with Webviews. The key idea of our approach is to model the possible effects of injected malicious JavaScript code at the Java level, avoiding to deal with JavaScript semantics. In particular, this allowed us to rely on state-of-the-art taint analysis for Android.

We implemented our approach in BabelView, and evaluated it on 11,648 applications, confirming its practical applicability and at the same time reporting on the state of Webview security in Android. With BabelView, we found 2,677 vulnerabilities in 1,663 applications, affecting more than 835 million users. We validated our results on a subset of applications and estimated that our tool achieves a precision of 90% at recall of 66%.

In future work, we plan to further refine the precision of our analysis by precisely modeling and tracking information flow in additional parts of the Android framework. We also plan to develop a dynamic analysis approach to automatic validation of BabelView's reported vulnerabilities.

Acknowledgments

We thank the CCS 2017 reviewers for their constructive feedback that helped to improve this paper. This work was in part supported by EPSRC grant EP/L022710/1 and a Google Faculty Award.

References

[1] P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna, "A Large-Scale Study of Mobile Web App Security," in *Proceedings of the Mobile Security Technologies Workshop (MoST)*, May 2015.

[2] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the Android system," in *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011.* ACM, 2011, pp. 343–352.

[3] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du, "Touchjacking attacks on web in android, ios, and windows phone," in *Foundations and Practice of Security - 5th International Symposium, FPS 2012, Montreal, QC, Canada, October 25-26, 2012, Revised Selected Papers*, ser. LNCS, vol. 7743. Springer, 2012, pp. 227–243.

[4] D. R. Thomas, "The lifetime of Android API vulnerabilities: Case study on the javascript-to-java interface (transcript of discussion)," in *Security Protocols XXIII - 23rd International Workshop, Cambridge, UK, March 31 - April 2, 2015, Revised Selected Papers*, ser. LNCS, vol. 9379. Springer, 2015, pp. 139–144.

[5] "WebKit," https://developer.android.com/reference/android/webkit/package-summary.html, May 2017.

[6] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben, "Why Eve and Mallory love Android: an analysis of Android SSL (in)security," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 2012, pp. 50–61.

[7] B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang, "Web-to-application injection attacks on android: Characterization and detection," in *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, ser. LNCS, vol. 9327. Springer, 2015, pp. 577–598.

[8] A. B. Bhavani, "Cross-site scripting attacks on Android webview," *CoRR*, vol. abs/1304.7451, 2013.

[9] MWR InfoSecurity, "WebView addJavascriptInterface Remote Code Execution," https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/, Sep. 2013.

[10] A. D. Brucker and M. Herzberg, "On the static analysis of hybrid mobile apps - A report on the state of apache cordova nation," in *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, ser. LNCS, vol. 9639. Springer, 2016, pp. 72–88.

[11] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for Android hybrid applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 2016, pp. 250–261.

[12] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 2014, pp. 259–269.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, 1999, p. 13.

[14] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing Android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014.

[15] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "Androzoo: collecting millions of Android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 468–471.

[16] P. Mutchler, Y. Safaei, A. Doupé, and J. C. Mitchell, "Target fragmentation in Android apps," in *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 204–213.

[17] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, "Measuring the declared SDK versions and their consistency with API calls in Android apps," in *Wireless Algorithms, Systems, and Applications - 12th International Conference, WASA 2017, Guilin, China, June 19-21, 2017, Proceedings*, 2017, pp. 678–690.

[18] D. R. Thomas, A. R. Beresford, and A. C. Rice, "Security metrics for the Android ecosystem," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2015, Denver, Colorado, USA, October 12, 2015*, 2015, pp. 87–98.

[19] "Android Intent Actions," https://developer.android.com/reference/android/content/Intent.html, Sep. 2017.

[20] S. Margaritelli, "BetterCAP," https://www.bettercap.org, Sep. 2016.

[21] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[22] 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, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 66–77.

[23] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: Webview exploitation," in *6th USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '13, Washington, D.C., USA, August 12, 2013*, 2013.

[24] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in Android applications," in *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, ser. LNCS, vol. 8267. Springer, 2013, pp. 138–159.

[25] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/mobile application frameworks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[26] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 104–115.

[27] M. Shehab and A. A. Jarrah, "Reducing attack surface on cordova-based hybrid mobile apps," in *Proceedings of the 2nd International Workshop on Mobile Development Lifecycle, MobileDeLi 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 2014, pp. 1–8.

[28] X. Jin, L. Wang, T. Luo, and W. Du, *Fine-Grained Access Control for HTML5-Based Mobile Applications in Android*. Cham: Springer International Publishing, 2015, pp. 309–318.