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

Claudio Rizzo, Lorenzo Cavallaro, and Johannes Kinder

Royal Holloway, University of London
{claudio.rizzo.2015|lorenzo.cavallaro|johannes.kinder}@rhul.ac.uk

Abstract. A Webview embeds a fully-fledged browser in a mobile application and allows that 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 custom 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 25,000 apps from various Android marketplaces, finding 10,808 potential vulnerabilities in 4,997 apps. Taken together, the apps reported as problematic have over 3 billion installations worldwide. We manually validate a random sample of 50 apps and estimate that our fully automated analysis achieves a precision of 81% at a recall of 89%.

Keywords: Webview · JavaScript interface · Injection · Static analysis

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 [17]. Cross-platform frameworks such as Apache Cordova, which allow apps to be written entirely in HTML and JavaScript, have contributed to this high rate of adoption and given rise to the notion of *hybrid applications*. Even otherwise native applications often embed Webviews for displaying login screens or additional web content.

Unfortunately, Webviews bring new security threats [15–17, 24]. While the Android Webview uses WebKit to render the page, the security model can be modified by app developers. Whereas 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 [7, 17].

The observation that exposed interfaces can pose a security risk was made in previous work [4, 9]; however, not all interfaces are dangerous or offer meaningful control to an attacker. The intuition is that flagging up—or even removing from the marketplace—any applications with an exposed JavaScript interface would be an excessive measure. By assessing the risk posed by an application, we can focus attention on the most dangerous cases and provide meaningful feedback to developers.

We rely on static analysis to evaluate the potential impact of an attack against Webviews, with respect to the nature of the JavaScript interfaces. Our key idea is that we can instrument an application 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.

Instrumenting the target application allows us to build on existing mature tools for Android flow analysis. This design makes our approach particularly robust, which is important on a quickly changing platform such as Android. In addition, since our instrumentation is over-approximate, we inherit any soundness guarantees offered by the flow analysis used. Independently of us, Yang et al. [31] developed a related approach to address the same problem, but with a closed source system relying on a custom static analysis. Our paper makes the following contributions:

– We introduce BabelView, a novel approach to evaluate the impact of code injection attacks against Webviews based on information flow analysis of applications instrumented with an attacker model. BabelView is implemented using Soot [27] and is available as open source.
– We analyze 25,000 applications from the Google Play Store to evaluate our approach and survey the current state of Webview security in Android. Our analysis reports 10,808 potential vulnerabilities in 4,997 apps, which together are reported to have more than 3 billion installations. We validate the results on a random sample of 50 applications and estimate the precision to be 81% with a recall of 89%, confirming the practical viability of our approach.

In the remainder of the paper, we briefly explain Android WebViews (§2) and introduce our approach (§3) before describing the details of our implementation (§4). We evaluate BabelView and report the results of our Android study (§5) and discuss limitations (§6). Finally, we present related work (§7) and conclude (§8).

2 Android WebViews

To provide the necessary context for the remainder of the paper, we first introduce key aspects of Android Webviews. An Android application can instantiate a Webview by calling its constructor or by declaring it in the Activity XML layout, from where the 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 into 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.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();
wView.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 `wView`. 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 to decide which functionality to expose in an interface that is more security-critical than it appears. It is easy for a developer to erroneously assume the JavaScript interface to be a trusted internal interface, 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 (see §3.1). 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.

[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 [19].

3 Overview

We now introduce our approach by laying out the attacker model (§3.1), describing our instrumentation-based model for information flow analysis (§3.2), and discussing 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. Indeed, any standalone browser that allows loading content via insecure HTTP has this vulnerability (while calling this a "vulnerability" may be controversial, it clearly has security implications and has led to an increasing adoption of HTTPS by default). The ubiquity of advertisement libraries in Android apps further increases the likelihood of foreign JavaScript code gaining access to JavaScript interfaces. Following this insight, we aim to pinpoint the risk of using a Webview that is embedded in an app. To do this, we assess the *degrees of freedom* an attacker gains from injecting code into a Webview with a JavaScript interface, which determines 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 actively try to inject JavaScript into the Webview—e.g., as man in the middle (see §5.5). We note, however, that other channels are available to manipulate the code loaded into a Webview, including malicious advertisements or site-specific cross-site-scripting attacks [4, 9, 10]. To abuse the JavaScript interface, the attacker then only requires the names of the interface methods, which can be obtained through reverse-engineering. Note that even a man in the middle becomes more powerful with access to the JavaScript interface: the interface can allow the attacker to manipulate and retrieve application and device data that would not normally be visible to the adversary. For instance, consider a remote access application with an interface method `getProperty(key)`, which retrieves the value mapped to a key in the application's properties. Without accessing the interface, an attacker may only ever observe calls to `getProperty` 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 aim to find potentially dangerous information flows from injected JavaScript into sensitive parts of the Java-based app and vice-versa. At first glance, this appears to require expensive cross-language static analysis, as recently proposed for hybrid apps [5, 13]. However, we can avoid analyzing JavaScript code because our attacker model assumes that the JavaScript code is controlled by the attacker. Therefore, we want to model the actions performed by *any possible JavaScript code*, and not that of developer-provided code that is supposed to execute in the Webview.

To this end, we 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 (we discuss alternative solutions

Algorithm 1 Information flow attacker model

```
1: procedure ATTACKER
2:     while true do
3:         choose iface ∈ JS-interfaces
4:         result ⟵ iface(source(), source(), . . . )
5:         sink(result)
```

in §6). We replace 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 [2] 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();
}
@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 helps to reduce false positives. In the BabelView, the loop in Algorithm 1 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 to perform a cross-language taint analysis and moves all reasoning into the Java domain. However, we must make sure

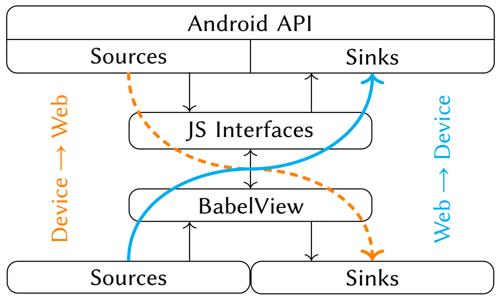

Fig. 1. BabelView models flows between the attacker and sensitive sources and sinks in the Android API that cross the JavaScript interface.

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 [2]. 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. Obviously, this is not necessarily true for other semantics: for example, the instrumented application would likely crash if it were executed on an emulator or real device.

4 BabelView

In this section, we explain the different phases of our analysis. Figure 2 provides a high-level overview: in Phase 1 (§4.1), we perform a static analysis to retrieve all interface objects and methods, and associate them to the respective Webviews. In Phase 2 (§4.2), we generate the BabelView, and, in Phase 3 (§4.3), we instrument the target application with it. In Phase 4 (§4.4), we run the taint flow analysis on the resulting applications and finally, in Phase 5 (§4.5), we analyze the results for flows involving the BabelView.

We implemented our static analysis and instrumentation using the Soot framework [27]; our information flow analysis relies on FlowDroid [2]. Overall, our system adds about 6,000 LoC 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. The goal of this stage is to compute a mapping from Webview classes to classes of interface objects that may be added to them at any point during execution of the app.

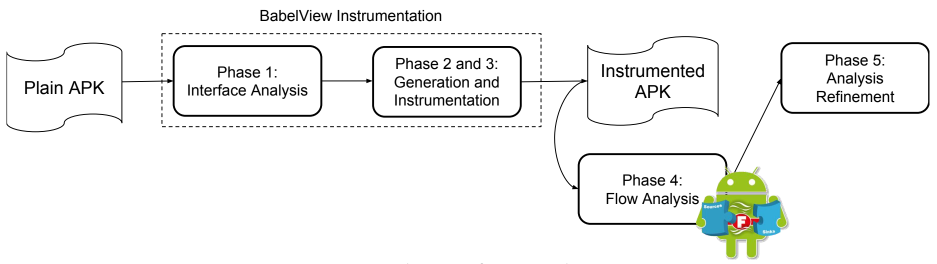
BabelView Instrumentation
Plain APK
Phase 1: Interface Analysis
Phase 2 and 3: Generation and Instrumentation
Instrumented APK
Phase 4: Flow Analysis
Phase 5: Analysis Refinement

Fig. 2. Phases of our analysis.

Using Soot, we can generate the application call graph and precisely resolve callers and callees. We iterate through all classes and methods, identifying all calls to `addJavascriptInterface`, from where we then extract Webviews that will hold interface objects. We make sure to treat inheritance and polymorphism soundly in this stage, e.g., where parent classes are used in variable declarations. We illustrate our approach to handle this on the following code example:

```
class FrameworkBridge {
  @JavascriptInterface
  public int foo() {...}
}

class MyBridge extends FrameworkBridge {
  @JavascriptInterface
  public int bar() {...}
}

class MyWebView extends WebView {...}

void initInterface(WebView aWebView, FrameworkBridge aBridge) {
  aWebView.addJavascriptInterface(aBridge, "Android");
}

...
MyWebView mWebView = new MyWebView();
initInterface(mWebView, new MyBridge());
...
```

The code is adding the interface `MyBridge` to `mWebView`, an instance of `MyWebView`. The method `initInterface` is a wrapper (say, from a hybrid app framework) that contains the actual call to `addJavascriptInterface`. When processing the call, we extract the types of `aWebView` and `aBridge` from their parameter declarations. For the Webview, we must process all descendants of its declared class to include the types of all possible instances. For `aWebView`, this means we must instrument all descendants (including anonymous classes) of `WebView`, i.e., `WebView` and `MyWebView`.

Similarly, we are interested in the type of aBridge. Again, we must iterate over all subclasses of its declared type FrameworkBridge to ensure capturing the bridge added at runtime. However, since addJavascriptInterface is of the unconstrained type Object, this could potentially include all classes. Therefore, we restrict processing to just those subclasses that contain at least one @JavascriptInterface annotation. As a result, we obtain a superset of all interface objects that can be added by this method, i.e., FrameworkBridge and MyBridge.

Continuing the example, we now have the mapping from Webview classes to classes of interface objects as $\{\texttt{WebView} \mapsto \{\texttt{FrameworkBridge}, \texttt{MyBridge}\}, \texttt{MyWebView} \mapsto \texttt{FrameworkBridge}, \texttt{MyBridge}\}\}$. Any additional occurrences of addJavascriptInterface will be processed analogously and the results added to the set. Because the analysis in this phase is conservative in collecting compatible types, the result is a sound over-approximation of the mapping of Webviews to JavaScript interfaces that can occur at runtime (modulo inaccuracies from dynamic code, see §6).

4.2 Phase 2: Generating the BabelView

We generate a BabelView class for each WebView in the mapping. Each BabelView defines 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 the 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. We automatically generate these methods as a call to their **super** implementation followed by a Java implementation of the attacker model, Algorithm 1. Finally, the BabelView is equipped with two stub methods, leak and taintSource, representing a tainted sink and a tainted input, respectively.

4.3 Phase 3: Instrumentation

In the next phase, we instrument the application to replace its Webviews with our generated BabelView instances. The instrumentation is case-dependent 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 uses to obtain the Webview instance (e.g., in order to add the JavaScript interface to it). To identify the calls to findViewById returning a Webview, our instrumenter identifies explicit casts to a Webview class. Because we do not parse the XML layout itself, we arbitrarily choose one of the constructors of the BabelView. 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 our approach relies on instrumenting the application under analysis, it is agnostic to the specific flow analysis. We decided to rely on the open source implementation of FlowDroid [2], inheriting its context-, flow-, field-, and object-sensitivity, as well as its life cycle-awareness.

Sources and sinks are selected corresponding to sensitive information sources and device functions, modified from the set provided by SuSi [22]. We further include the sources and sinks used in the BabelView classes.

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, information flows indicating that sensitive functionality is exposed via the JavaScript interface are identified, triggering an alarm showing a potential vulnerability. For instance, consider an `Intent` object initialized to perform phone calls. A flow from `source` to `putExtra` will taint the `Intent`; if it is then passed as an input to `startActivity`, an attacker can perform calls on behalf of the user.

4.5 Phase 5: Analysis Refinement

Preferences. Taint analysis cannot distinguish between individual key-value pairs in a map. `Preferences` are a commonly used map type in Android apps that often store sensitive information as a key-value pair. After the information flow analysis, we refine our results by statically deriving values of keys for access to preferences. Our definition of sources and sinks allows to identify both flows from and to the `Preferences`. Given two flows, one inserting and the other retrieving values from `Preferences`, we are interested in understanding whether (i) the value is of the same type and (ii) the access key is the same. If these conditions are met, we have identified a potential leak via Preferences. To determine the key values, we modeled `StringBuilder` and implemented an intra-procedural constant propagation and folding for strings. Finally, if an interface method allows web content to interact with a preferences object, BabelView reports all keys used to access it, since preferences can be used to store sensitive values. This allows to inspect flows to or from preferences entries, even if these values are not dependent on a specific source in the Android API. We match key names against a list of suspicious entries, which can highlight potential leaks of sensitive app-specific information (see §5.7). In the same manner, we also highlight suspiciously named interface methods.

Intents. Flow analysis can detect situations where Intent creation depends on tainted input. However, it tells nothing about the type of the Intent created, as this depends on specific parameters, e.g, those provided to its `setAction` method. For interpreting results, it is important, however, to know 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`. 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 interface method; i.e., the interface method that triggered the flow must be reachable.

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 research questions to evaluate our approach:

1. **Can BabelView successfully process real-world applications?** We conduct a study on a randomly selected set of applications from the AndroZoo [1] dataset and provide a breakdown of all results (§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, estimating 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 alarms?** We report results per alarm, which provides an insight into the prevalence of potential vulnerabilities (§5.3).
5. **Are there types of potential vulnerabilities that are likely to occur in combination?** We compute the correlation between alarms raised by our analysis and analyze our findings (§5.6).

Unfortunately, we were unable to conduct a direct comparison with BridgeScope, the work most closely related to ours. Despite helpful communication, the authors were ultimately unable to share neither their experimental data nor their implementation with us. In the spirit of open data, we make all our code and data available[2].

5.1 Methodology

We obtained our dataset from AndroZoo [1], using the list of applications available on July 22nd, 2016, when it contained about 4.4 million samples. We downloaded a random subset of 209,069 apps, and then filtered our dataset for applications containing a Webview, a call to `addJavascriptInterface`, and granting permission to access the Internet. As a result, we obtained 62,674 total applications. Finally, from the obtained sample, we randomly extracted 25,000 applications found in the Google Play Store, which we used for our analysis.

We ran BabelView on five servers: one 32-core with 250GiB of RAM and four 16-core with 125GiB of RAM. Each application took on average 180 seconds to complete. The high precision of FlowDroid's information flow analysis can lead to long processing time in the order of hours. Therefore, we set a time limit of 15 minutes, which was a sweet spot in the sense that apps taking longer would often go over an hour. A positive

[2] `https://github.com/ClaudioRizzo/BabelView`

effect of our instrumentation-based approach is that we benefit from improvements in the underlining flow analysis. Indeed, over the duration of this project, we saw a noticeable accuracy enhancement from the constant improvements on FlowDroid.

Each application underwent three main phases: (i) BabelView instrumentation, (ii) FlowDroid analysis on the instrumented app and (iii) analysis of the resulting flows to identify suspect flows and raise alarms. On the reported applications, we performed a feasibility analysis. We searched the app for plain `http://` URLs and assess the resilience of the app against injection attacks.

5.2 Applicability

Running our tool chain on the 25,000 target applications resulted in 1,286 general errors and 3,837 flow analysis timeouts. The remaining 19,877 apps were successfully analyzed and we obtained the following breakdown: 832 applications had no interface objects at all or no interface methods in case the target API was version 17 or above; 14,048 applications had no flows involving our attacker model; and 4,997 were reported as dangerous, i.e., containing flows due to the attacker behavior. This amounts to a rate of 26.2%. We investigated the reasons for the crashes, and most happened either due to unexpected byte code that Soot fails to handle or while FlowDroid's taint analysis was computing callbacks.

Among applications with interface objects, we also considered those targeting outdated versions of the Android API, since this is still a common occurrence [18,25,28]. When using Webviews prior to API 17, any app is trivially vulnerable to an arbitrary code execution disclosed in 2013[3]. Despite targeting an old API version, if compiled with a newer Android SDK, these applications can still use the `@JavascriptInterface` annotation. While the annotation itself does not provide extra security, these apps may target newer APIs in future releases [24].

5.3 Alarms Triggered

We successfully used BabelView to examine 19,877 applications. We found that 4,997 of them triggered an alarm (i.e., our analysis reported a potential vulnerability), meaning that the interface methods could be exploited by foreign JavaScript from injection or advertisement. Table 1 shows a breakdown of all the alarms we observed in our analysis. Among the most common alarms, we observed the possibility of writing to the File System (Write File), capability to start new applications (Start App), violation of the Same Origin Policy (Frame Confusion) and the possibility of exploiting the old reflection attack due to Android API prior to v17.

Writing File capabilities show the developers' need for storing app-external data usually coming from an app-dedicated server. We also observed that many applications implement advertising libraries which need to open a new application, usually Google Play Store, to allow the user to download or visualize some information. Unfortunately, the package name of the application to open is given as input to an interface method, enabling a possible attacker to control which app to start. Same-Origin-Policy violations are also very common: this is the case when a `loadUrl` is invoked with input from the interface methods, controlling what is loaded in to a frame. As described by Luo

[3] https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/

Alarm	#Apps	Alarm	#Apps	Alarm	#Apps
Open File	385	Write File	1,444	Read File	593
TM Leaks	39	Pref. TM Leaks	4	Pref. Connectivity Leaks	4
SQL-lite Leaks	136	SQL-lite Query	438	Pref. SQL-lite Leaks	11
GPS Leaks	43	Pref. GPS Leaks	1	Directly Send SMS	6
Directly Make Calls	19	Call via Intent	314	Email/SMS via Intent	778
Take Picture	7	Download Photo	317	Play Video/Audio	378
Edit Calendar	357	Post to Social	293	Start App	1,321
API prior to 17	1,039	Unknown Intent	1,107	Frame Confusion	1,039
Fetch Class	85	Fetch Constructor	0	Constructor init	13
Fetch Method	85	Method Parameter	622		

Table 1. Number of applications per alarm category. Pref. stands for indirect leaks via a Preference object; TM stands for Telephony Manager.

et al. [15], JavaScript executing in an iframe runs in the context of the main frame, violating the SOP.

Many applications still target an API version prior to 17 [18, 25, 28], often due to backwards compatibility or simply due to confusion in declaring the SDK version. Other alarms involve the possibility to prompt the user with an email or a text message to send, directly sending an SMS or performing a phone call; prompting the user with the call dialer; posting content to social network; interacting with the calendar by creating or editing an event; playing videos or audio; leaking sensitive information like the device ID or phone numbers (i.e. TM Leaks), GPS position, SQL information, etc.

Finally, we shed light on the possible use of Java Reflection inside interface methods. Fetch Class, Fetch Constructor, Constructor init, Fetch Method and Method Parameter are all signs that an attacker controls input used to execute methods via Java reflection. Although these are rare situations and often hard to exploit, they are extremely high reward for an attacker as they can potentially allow to circumvent the `@JavascriptInterface` annotation, leading to arbitrary code execution. We manually analyzed some applications presenting these alarms and in some cases an attacker could take control of a method and its parameters, leading to remote code execution.

5.4 Manual Validation

We used manual validation to estimate the accuracy of our analysis. In particular we sampled and manually analyzed (i.e., reversed and decompiled) 50 applications. We evaluated two aspects:

1. How accurate is BabelView with respect to each individual alarm it raises?
2. Does BabelView function as an effective alarm system for hybrid apps?

We began checking all types of alarms for each app and we established whether an alarm was correctly triggered or correctly not triggered. We observed 42 TPs (True Positives), 10 FPs (False Positives), 1,494 TNs (True Negatives) and 5 FNs (False Negatives). From this, we can compute a precision of 81% and a recall of 89% for our analysis.

The results obtained are in line with our expectations. Our instrumentation does not alter the semantics of applications other than adding a model of attack behavior. Therefore, our precision depends on the underlining flow analysis. However, more false positives could be introduced due to the object-insensitivity of our instrumentation—i.e., we distinguish types but not instances of Webviews. Similarly, a very low false negative rate is common for data flow analysis; however, FNs are still possible, mainly due to incomplete Android framework.

To evaluate BabelView on a per-app basis, we consider a true positive the case where an app contains at least one potential vulnerability and at least one alarm is raised. True negatives and false positives/negatives follow accordingly. We observed 19 TPs, 2 FPs, 29 TNs, and 0 FNs, which yields a precision of 90% and a recall of 100%. This suggest that BabelView performs well as an alarm system for potentially dangerous applications. Even if individual alarms can be false positives, the correlation of dangerous interfaces appears to leads to highlighted apps being problematic with high probability. The false negatives that are present when taken per vulnerability disappear when analyzed on a per app basis.

5.5 Feasibility Analysis

To better understand the feasibility of exploiting potential vulnerabilities highlighted by BabelView, we measured the difficulty of performing an injection attack. To this end we use a three-step process: (i) we check the application for TLS misuse using MalloDroid [7]; (ii) we search for hard-coded URLs beginning with `http://`, suggesting that web content could be loaded via an insecure channel; and (iii) we actively injected JavaScript code into Webviews.

MalloDroid reported 61.5% of applications using TLS insecurely and 98.7% of apps were found hard-coding HTTP URLs. In order to actively inject JavaScript, we stimulated each reported application with 100 Monkey[4] events and actively intercepted the connection (using Bettercap[5]), trying to execute a JavaScript payload. Moreover, we set up our own certificate authority and also tried SSL strip attacks. The goal of the injection was to determine whether the reported interface methods were present in the Webview. To this end, we generated JavaScript code checking for the presence of the interface objects reported by the BabelView analysis. We were able to inject JavaScript in 1,275 applications and in 482 cases we confirmed the presence of the vulnerable interface object.

5.6 Correlation of Alarms

We were interested in finding correlations among the alarm categories we identified. This does not only account for common patterns of functionality, but also identifies single alarms that taken together could increase the attack capabilities, e.g., combining opening and writing of a file results in writing of arbitrary files.

We can see in the correlation matrix in Figure 3 that alarms involving related functionality tend to be positively correlated (in red). For example, opening and writing a file; SQL queries and leaks; and operations involving intents such as call via intent, send email, edit calendar, play video, post to social, and download pictures. While some

[4] `https://developer.android.com/studio/test/monkey.html`

[5] `https://www.bettercap.org`

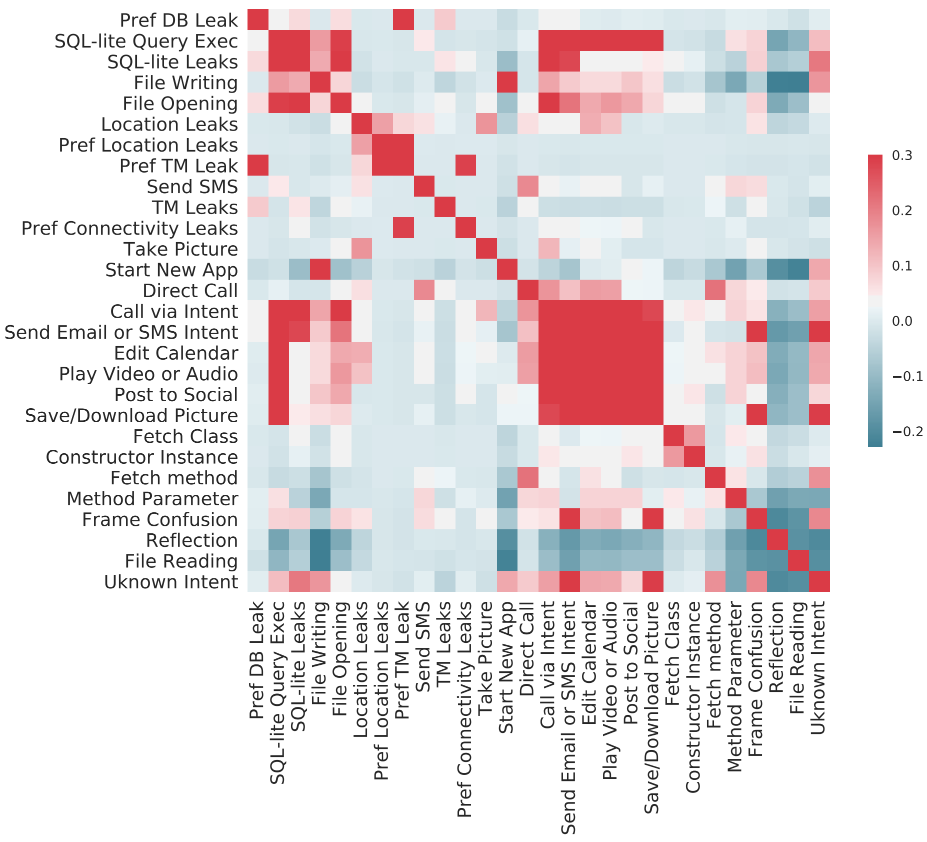

Fig. 3. Correlation matrix of alarms.

correlations are evident, some appear incidental, such as intent calls and playing of videos. Based on manual inspection (see §5.7), we found that these categories of alarms often appear together in apps using common libraries, e.g., for advertisements.

5.7 Individual Case Studies

We now report individual case studies to illustrate the nature of our findings.

Advertising Libraries. During the evaluation, we discovered an advertising library, used by 353 of 4,997 applications, which implements a Webview exposing many sensitive interface methods. In particular, a successful JavaScript injection would allow an attacker to perform different actions, including downloading/saving of pictures, sending email or SMS by manipulating `Intents`, playing audio or videos on the victim's phone, opening new applications, creating calendar events, and posting to social networks.

Another library, used by 1,507 applications, allows an attacker to start new applications on the phone, controlling the `Intent` extras provided to the `Activity`.

Game App. Among our results, we found a game application ("SwingAid Level up Golf") that uses several Webviews and JavaScript interfaces leading to different alarms: SQL-lite leaks via preferences, frame confusion, and telephony manager Leaks. Moreover, we discovered the value *loginPwd* among preferences keys accessible from a JavaScript interface. We were able to manually confirm all alarms as true positives. Interface methods accessible when creating an account creation within the game include `getAccountEmail`, `getPhoneNumber`, and `getUserPwd`. We successfully performed a man-in-the-middle attack and injected JavaScript to access all three methods. The account e-email and phone number are accessible immediately upon attempting to create an account. The password is stored in a local database, cached in the preferences and accessible with the *loginPwd* key. When the user visits the account creation page a second time, the password can be stolen via the interface method.

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 extract it directly. Since our discovery, all issues have been resolved in a newer version of the application (version 2.6).

6 Limitations and Discussion

Avoiding Instrumentation In principle, we could avoid instrumenting the application by summarizing interface methods with an interprocedural taint analysis. However, to achieve the same precision, the analysis would have to be computationally expensive: on method entry, any reachable field in any reachable object (not just arguments of the interface method) would have to be treated as carrying individual taint. On method exit, the effects on all reachable fields would have to stored, before resolving the effects among all interface method summaries. Our instrumentation-based approach not only avoids this cost, but also allows us to factor out flow analysis into a separate tool, a design choice that improves robustness and maintainability.

Analysis Limitations Our system, together with the underlying flow analysis, is subject to common limitations of static analysis and hence can fail to detect Webviews and interfaces instantiated via native code, reflection, or dynamic code loading. In principle, this currently allows a developer intent on doing so to hide sensitive JavaScript APIs. However, we focus on benign software and vulnerabilities that are honest mistakes rather than planted backdoors. Still, we note that BabelView would automatically benefit from future flow analyses that may counteract evasion techniques.

A potential source of false positives is that BabelView does not distinguish Webview instances of the same type and will conservatively join the JavaScript interfaces of all instances. Furthermore, our analysis loses precision 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 different treatment of this would be a potential source of false negatives, since preferences persist across application restarts.

Attack Feasibility In our feasibility analysis, we actively try to inject JavaScript code into a Webview, aiming at identifying whether the reported interface object is present in the Webview. The presence of the interface object means that all its interface methods are available to use, including the one BabelView reported. However, we do not actively invoke these methods and thus we cannot be sure of their exploitability.

Mitigating Potential Vulnerabilities To avoid giving potential attackers control over sensitive data and functionality, developers can follow a set of design principles. First of all, Webview contents should be exclusively loaded via a secure channel. Second, as mentioned in the Android developer documentation, Webviews should only load trusted contents. External links have to be opened with the default browser. For also protecting against malicious ads or cross-site-scripting attacks, JavaScript interfaces should offer an absolute minimum of functionality and avoid arguments as far as possible. Finally, recent work also introduced novel mechanisms to enforce policies on hybrid applications (see §7.2).

7 Related Work

We now review work on vulnerabilities and attacks against Webview (§7.1), discuss related work on policies and access control (§7.2), and contrast with work on instrumentation-based modeling (§7.3).

7.1 Webview: Attacks and Vulnerabilities

Webview vulnerabilities have been widely studied [4, 6, 15–17, 20]. Luo et al. give a detailed overview of several classes of attacks against Webviews [15], providing a basis for our work. Neugschwandtner et al. [20] 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 of this paper.

A step forward towards this was made by Bifocals [6], a static analysis tool able to identify and evaluate vulnerabilities in Webviews. Bifocals looks for potential Webview vulnerabilities (using JavaScript interfaces and loading third party 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 pose an (application-specific) risk without making use of permissions. At the same time, 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.

The means by which malicious code can be injected into the Webview have been discussed in previous work [9, 10]. Having to interact with many forms of entities, HTML5-based hybrid applications expose a broader surface of attack, introducing new vectors of injection for cross-site-scripting attacks [10]. While these attacks require the user to directly visit the malicious page within the Webview, Web-to-Application injection attacks (W2AI) rely on intent hyperlinks to render the payload simply by clinking a link in the default browser [9]. 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.

A large scale study on mobile web applications and their vulnerabilities was presented by Mutchler et al. [17], but did not study the nature of the exposed JavaScript interfaces. Li et al. [14] studied a new category of fishing attacks called *Cross-App WebView infection*. This new type of attacks exploits the possibility of issuing navigation requests from one app's Webview to another via Intent deep linking and other URL schemata. This can trigger a chain of requests to a set of infected apps.

Most closely related to our work is the concurrently developed *BridgeScope* [31], a tool to assess JavaScript interfaces based on a custom static analysis. Similar to our work, BridgeScope allows to detect potential flows to and from interface methods. BridgeScope uses a custom flow analysis, whereas our approach intentionally allows to reuse state-of-art flow analysis tools. While BridgeScope's flow analysis performs well on benchmarks, there appears to be no specific treatment of Map-like objects such as `Preferences` of `Bundle`.

In recent work, Yang et al. [29] have combined the information of a deep static analysis with a selective symbolic execution to actively exploit event handlers in Android hybrid applications. In *OSV-Hunter* [30], they introduce a new approach to detect Origin Stripping Vulnerabilities. These type of vulnerabilities persist when upon invocation of `window.postMessage`, it is not possible to distinguish the identity of the message sender or even safely obtain the source origin. This is inherently true for Hybrid applications, where developers often rely on JavaScript interfaces to fill the gap between web and the native platform.

7.2 Webview Access Control

There have been several proposals to bring origin-based access control to Webviews [8, 23, 26]. Shehab et al. [23] 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 [8], 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 white-listed, it can access any resource exposed. Jin et al. [11] propose a fine-granular solution in a system that allows developers to assign different permissions to different frames in the Webview.

Tuncay et al. [26] increase granularity further in their Draco system. 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. Another framework allowing developers to define security policies is HybridGuard [21]. Differently from Draco, HybridGuard has been entirely developed in JavaScript, making it platform independent and easy to deploy on different platform and hybrid development framework. Both Draco and HybridGuard could provide an interesting solution to the problem of securing an interface BabelView is rising an alarm for, without unduly restricting its functionality.

7.3 Instrumentation-based Modeling

Synthesizing code to trigger specific function interfaces is not a new problem and traces back to generating verification harnesses, e.g., for software model checking [3, 12]. On Android, FlowDroid [2] uses a model that invokes callbacks in a "dummy main" method, taking into account the life cycle of Android activities. While the problems share some similarity, JavaScript interfaces and Webviews are inherently varied and app-specific. Therefore, we require a static analysis and cannot rely on fixed signatures. Furthermore, because our model represents an attacker instead of a well-defined system, calls can appear out of context anytime web content can be loaded in the Webview, i.e., after a `loadUrl`-like method.

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, thereby avoiding any direct reasoning about JavaScript semantics. In particular, this allowed us to rely on a robust state-of-the-art implementation of taint analysis for Android.

We implemented our approach in BabelView, and evaluated it on 25,000 applications, confirming its practical applicability and at the same time reporting on the state of Webview security in Android. With BabelView, we found 10,808 potential vulnerabilities in 4,997 applications, affecting more than 3 billion users. We validated our results on a subset of applications where we achieved a precision of 81% at a recall of 89% when measured per alarm, or a precision of 90% and a recall of 100% when measured per application.

Acknowledgments

We would like to thank our shepherd, Angelos Stavrou, and the anonymous reviewers for their valuable feedback. We are grateful to Roberto Jordaney, Blake Loring, Duncan Mitchell, James Patrick-Evans, Feargus Pendlebury, and Versha Prakash for their help and their comments on earlier drafts of this paper. This work was in part supported by EPSRC grant EP/L022710/1 and a Google Faculty Award.

References

1. K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. Androzoo: collecting millions of Android apps for the research community. In *Proc. 13th Int. Conf. Mining Software Repositories (MSR)*, pages 468–471. ACM, 2016.
2. 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 Conf. Programming Language Design and Implementation (PLDI)*, pages 259–269. ACM, 2014.
3. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proc. 2006 EuroSys Conf.*, pages 73–85. ACM, 2006.
4. A. B. Bhavani. Cross-site scripting attacks on Android WebView. *CoRR*, abs/1304.7451, 2013.

5. A. D. Brucker and M. Herzberg. On the static analysis of hybrid mobile apps - a report on the state of Apache Cordova nation. In *Proc. Int. Symp. Engineering Secure Software and Systems (ESSoS)*, pages 72–88. Springer, 2016.
6. E. Chin and D. Wagner. Bifocals: Analyzing WebView vulnerabilities in Android applications. In *Int. Workshop Information Security Applications (WISA)*, pages 138–159. Springer, 2013.
7. 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 *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, pages 50–61. ACM, 2012.
8. M. Georgiev, S. Jana, and V. Shmatikov. Breaking and fixing origin-based access control in hybrid web/mobile application frameworks. In *Annu. Network and Distributed System Security Symp. (NDSS)*, 2014.
9. B. Hassanshahi, Y. Jia, R. H. C. Yap, P. Saxena, and Z. Liang. Web-to-application injection attacks on Android: Characterization and detection. In *European Symp. Research in Computer Security (ESORICS)*, pages 577–598. Springer, 2015.
10. 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 *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, pages 66–77. ACM, 2014.
11. X. Jin, L. Wang, T. Luo, and W. Du. Fine-grained access control for html5-based mobile applications in Android. In *Int. Information Security Conf. (ISC)*, pages 309–318, 2013.
12. J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Proc. 10th Int. Conf. Formal Methods in Computer-Aided Design (FMCAD)*, pages 43–50, 2010.
13. S. Lee, J. Dolby, and S. Ryu. Hybridroid: static analysis framework for Android hybrid applications. In *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE)*, pages 250–261. ACM, 2016.
14. T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han. Unleashing the walking dead: Understanding cross-app remote infections on mobile WebViews. In *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS).* ACM, 2017.
15. T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on WebView in the Android system. In *Annu. Computer Security Applications Conference (ACSAC)*, pages 343–352. ACM, 2011.
16. T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking attacks on web in Android, iOS, and Windows phone. In *Int. Symp. Foundations and Practice of Security (FPS)*, pages 227–243. Springer, 2012.
17. P. Mutchler, A. Doupé, J. Mitchell, C. Kruegel, and G. Vigna. A large-scale study of mobile web app security. In *Proc. IEEE Symp. Security and Privacy Workshops (SPW), Mobile Security Technologies (MoST).* IEEE, 2015.
18. P. Mutchler, Y. Safaei, A. Doupé, and J. C. Mitchell. Target fragmentation in Android apps. In *Proc. IEEE Symp. Security and Privacy Workshops (SPW), Mobile Security Technologies (MoST)*, pages 204–213. IEEE, 2016.
19. MWR InfoSecurity. WebView addJavascriptInterface remote code execution. `https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/`, Sept. 2013.
20. M. Neugschwandtner, M. Lindorfer, and C. Platzer. A view to a kill: WebView exploitation. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2013.
21. P. H. Phung, A. Mohanty, R. Rachapalli, and M. Sridhar. HybridGuard: A principal-based permission and fine-grained policy enforcement framework for web-based mobile applications. In *Proc. IEEE Symp. Security and Privacy Workshops (SPW), Mobile Security Technologies (MoST)*, 2017.
22. S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *Annu. Network and Distributed System Security Symposium (NDSS).* The Internet Society, 2014.

23. M. Shehab and A. A. Jarrah. Reducing attack surface on Cordova-based hybrid mobile apps. In *Proc. Int. Workshop on Mobile Development Lifecycle (MobileDeLi)*, pages 1–8. ACM, 2014.
24. D. R. Thomas. The lifetime of Android API vulnerabilities: Case study on the javascript-to-java interface (transcript of discussion). In *Int. Workshop Security Protocols*, pages 139–144. Springer, 2015.
25. D. R. Thomas, A. R. Beresford, and A. C. Rice. Security metrics for the Android ecosystem. In *Proc. ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 87–98. ACM, 2015.
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 *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, pages 104–115. ACM, 2016.
27. R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. Conf. Centre for Advanced Studies on Collaborative Research (CASCON)*, page 13, 1999.
28. 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 *Proc. Int. Conf. Wireless Algorithms, Systems, and Applications (WASA)*, pages 678–690, 2017.
29. G. Yang, J. Huang, and G. Gu. Automated generation of event-oriented exploits in Android hybrid apps. In *Annu. Network and Distributed System Security Symp. (NDSS)*, 2018.
30. G. Yang, J. Huang, G. Gu, and A. Mendoza. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *Proc. IEEE Symp. Security and Privacy (S&P)*, 2018.
31. G. Yang, A. Mendoza, J. Zhang, and G. Gu. Precisely and scalably vetting javascript bridge in Android hybrid apps. In *Int. Symp. Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2017.