LUDroid: A Large Scale Analysis of Android – Web Hybridization

1st Abhishek Tiwari
Software Engineering Group
University of Potsdam
Potsdam, Germany
tiwari@uni-potsdam.de

2nd Jyoti Prakash
Software Engineering Group
University of Potsdam
Potsdam, Germany
jyoti@uni-potsdam.de

3rd Sascha Groß
Software Engineering Group
University of Potsdam
Potsdam, Germany
saschagross@uni-potsdam.de

4th Christian Hammer
Software Engineering Group
University of Potsdam
Potsdam, Germany
chrhammer@uni-potsdam.de

***Abstract*—Many Android applications embed webpages via WebView components and execute JavaScript code within Android. Hybrid applications leverage dedicated APIs to load a resource and render it in WebView. Furthermore, Android objects can be shared with the JavaScript world. However, bridging the interfaces of the Android and JavaScript world might also incur severe security threats: Potentially untrusted webpages and their JavaScript might interfere with the Android environment and its access to native features.**

No general analysis is currently available to assess the implications of such hybrid apps bridging the two worlds. To understand the semantics and effects of hybrid apps, we perform a large-scale study on the usage of the hybridization APIs in the wild. We analyze and categorize the parameters to hybridization APIs for 7,500 randomly selected applications from the Google Playstore. Our results advance the general understanding of hybrid applications, as well as implications for potential program analyses, and the current security situation: We discover 6,375 flows of sensitive data from Android to JavaScript, out of which 82% could flow to potentially untrustworthy code. Our analysis identified 365 web pages embedding vulnerabilities and we exemplarily exploit them. Additionally, we discover 653 applications in which potentially untrusted Javascript code may interfere with (trusted) Android objects.

***Index Terms*—Android Hybrid Apps, Static Analysis, Information Flow Control**

I. Introduction

The usage of mobile devices is rapidly growing with Android being the most prevalent mobile operating system (global market share of 72.23% as of November 2018 [1]). Various reports [2], [3] reveal that the mobile application (app) usage is growing by 6% year-over-year and users are preferring mobile apps over desktop apps.

Considering these statistics, industry prioritizes mobile app development [4]. However, apps need to be developed for various platforms, such as Android and iOS, resulting in increased production time and cost. Traditional approaches require creation of a native application for each platform or of a universal web app. The former approach incurs redundant programming efforts, whereas, the latter suffers from the inability to access platform-specific information.

Hybrid mobile apps combine native components with web components into a single mobile application. Intuitively, hybrid apps are native applications combined with web technologies such as HTML, Javascript and CSS. On Android, a *WebView* [5] component, a chromeless browser [6] capable of displaying webpages, embeds the web applications into a view of the Android app. App Trends' developer survey [7] shows the increasing prevalence of hybrid applications. In the previous two years (2015-17), app development with native tools decreased significantly (by almost 7×), whereas the number of hybrid apps was growing in share of overall app development. By the end of 2019, 32.7% of the surveyed developers expect to completely abandon native development in favor of hybrid.

Due to the fact that hybrid apps combine native and web technologies in a single app, the attack surface for malicious activities increases significantly, as potentially untrusted code loaded at runtime, can interfere with the trusted Android environment. In our study with 7,500 random applications from the Google Play Store, we find that 68% of these apps use at least one instance of WebView and 87.9% of these install an active communication channel between Android and JavaScript. This includes exchange of various pieces of sensitive information, such as the user's location. To assess the impact on user privacy a standalone analysis of the Android or Javascript side is thus clearly insufficient. However, very limited work towards linking the assessment of both worlds can be found in the literature. Initial approaches either focus on type errors during hybrid communication [8] and/or only consider a very specific vulnerability arising of hybridization [9]–[15].

Hybrid communication leverages APIs like the *loadUrl* or *evaluteJavascript* methods, which, from inside an Android application, can either load a webpage into the WebView or execute Javascript code directly. To improve comprehension of hybrid communication we propose LUDroid, a framework supporting our semi-automatic analysis of hybrid communication on Android. We extract the information flows from Android to hybridization APIs and thus to the JavaScript engine to be executed in the displayed web page (if any), and categorize these flows into benign and transmitting sensitive data. We discover 6,375 sensitive flows from Android to Javascript.

The major parameter passed to *loadUrl* is the URL to be loaded. We analyze the syntax and semantics of each URL and provide a detailed categorization. As a byproduct, we find several vulnerabilities concerning the usage of these URLs. We successfully exploit some of these to demonstrate the threats. Alternatively, *loadUrl* and *evaluteJavascript* accept raw

Javascript code as parameter. We encountered code that loads additional JavaScript libraries into WebView. Unexpectedly, we also discovered 653 applications (with potentially untrusted Javascript code) transmitting data back to the Android bridge object. Therefore these apps implement two-way communication that may jeopardize the integrity of the Android environment, particularly as most external JavaScript is loaded without https and is thus prone to man-in-the-middle attacks. We found that state of the WebView is saved to Android before destruction of that component, transmission of identifying information used for advertisement or monetization purposes, as well as obfuscated code to preclude analysis of the semantics. We discuss the impact of our findings on potential program analyses that are to automatically identify issues of hybrid apps while taking hybrid communication concisely into account.

Technically we provide the following contribution:

- *Information Flow Analysis* We thoroughly investigate 7,500 real-world Android apps. We provide statistics on the information flows between Android and Javascript and identify leakage of sensitive information to the WebView.
- *URL Analysis* We perform an extensive analysis for the URLs used with *loadURL*, extracting various features. As a byproduct we identify applications that are vulnerable due to unencrypted transport protocols and exemplify the simplicity of a phishing attack.
- *Javascript analysis* We inspect the Javascript code passed to the hybridization APIs and identify a set of 73 distinct code snippets, most of which originating from third-part libraries. We extract relevant features and highlight their implications on program analysis of hybrid apps.

II. Background

A. Hybrid applications

A general disadvantage of native applications is that they are bound to a specific platform. For instance an Android application is bound to the Android platform and cannot easily be transformed into an iOS application. A developer wanting to support multiple platforms needs to implement a native application for each of these platforms separately, multiplying the implementation effort. Alternatively web applications execute in an arbitrary web browser and are therefore platform independent. However, they are restricted by a browser sandbox with very limited access to the native APIs of the mobile device. *Hybrid applications* have been proposed as a remedy, as they take full advantage of both approaches. They make extensive use of web requests, e.g., to display user interfaces. While having access to all native API methods granted by the Android permission system, development effort is reduced, as the user interface and its controllers can be retrieved via web requests and therefore do not need to be re-implemented.

B. WebView and loadURL

Hybrid applications on Android leverage *WebView*s. WebViews are user interface components that display webpages (without any browser bars), and thus provide a means to implement user interfaces as web pages instead of natively. The WebView class provides a *loadURL* method, which loads a webpage or executes raw Javascript. This method comes in two variants: *loadUrl(String url)* and *loadUrl(String url, Map<String,String> additionalHttpHeaders)*. While the first variant only takes a URL as argument, the second variant can additionally be passed HTTP headers for the request. Similar to a browser's location bar, one of the following parameters can be passed to *loadUrl*: (1) a remote URL leveraging protocols such as HTTP(S), (2) a local URL specified with protocol *file*, or (3) Javascript code prepended by the string *javascript:*. *WebView* leverages the appropriate renderer for each URL type automatically. Finally, a dedicated API *evaluateJavascript* executes JavaScript code directly.

III. Motivating example

In this section we will describe a simple hybrid Android example program (Listing 1 and 2) together with the communication between Android and the WebView component. We will then motivate the rationale behind our large-scale study to understand various factors concerning the usage of *WebView*s in realistic apps.

In Listing 1, a *WebView* object *myWebView* is retrieved from the Activity's UI via an identifier (line 2). By default execution of JavaScript in a *WebView* object is disabled but can be enabled by overriding the WebView's default settings (line 5). *WebView* provides the means to create a Java interface object that is shared with the WebView and can be accessed via JavaScript. Thus, the Android app's capabilities can be bridged to the Web component via *bridge communication* to, e.g., provide access to various sensors' data. In our example, an object of the class *Leaker* (see Listing 2) is shared (Listing 1, line 7) with the *WebView* object *myWebView* such that every webpage loaded into *myWebView* can use this object via the global variable *"Android"* (i.e. "Android" becomes a persistent property of the DOM's global object). Finally, the method *loadUrl* can be used in two ways: (1) to invoke JavaScript code directly (prepending a *javascript:* tag to the passed code) from Android, and (2) to load a custom URL (line 10) (which again may execute Javascript code specified in the web page).

Some previous work [8] take a first step into analyzing the data flows from Android to JavaScript but it is far from sound and mostly concentrates on potential type errors when passing data between the two worlds. To better understand which data flows are to be considered when analyzing an app consisting of a combination of Android and JavaScript code a thorough understanding of the methods *addJavascriptInterface* and *loadUrl* is required. In particular we are interested in the uses and potential abuses of this interface in the wild and their implications on the design of a program analysis for hybrid apps.

Consider line 9 in Listing 1, which reveals that the *loadUrl* method is invoking the *showToast* method defined in the *Leaker* class (Listing 2, line 3). This Java method retrieves the Android device's unique ID and returns it to the JavaScript code. Similar Javascript code could also be invoked in the loaded webpage (Listing 1, line 11) where it might be leaked

Listing 1: MainActivity.java

```
protected void onCreate(Bundle savesInstanceState) {
  WebView myWebView = (WebView)
      findViewById(R.id.webview);
  WebSettings webSettings = myWebView.getSettings();
// enable JavaScript on WebView
  webSettings.setJavaScriptEnabled(true);
  /* add interface object of type Leaker to the WebView's
      DOM as a property named "Android" of the global object
      */
  myWebView.addJavascriptInterface(new Leaker(this),
      "Android");
  // case 1: invoke Javascript from Android
  myWebView.loadUrl("javascript:"+"
      print(Android.showToast('Hello World'))");
  /* case 2: load a webpage (potentially executing
      JavaScript), the object "Android" persists as property
      of the DOM's global object */
  myWebView.loadUrl( "http://www.dummypage.com");
}
```

Listing 2: Leaker.java

```
// Add a JavascriptInterface annotation before the method you
      want to bridge
@JavascriptInterface
public String showToast(String toast) {
  TelephonyManager tManager = (TelephonyManager)
      mContext.getSystemService(Context.
      TELEPHONY_SERVICE);
  String uid = tManager.getDeviceId(); // get the device ID
  return uid;
}
```

to some untrusted web server together with more information the user enters into the web page. Note that state-of-the-art information flow analyses for Android cannot report these flows as they have no information whether the WebView's code actually leaks the shared data (even worse, many do not even consider *loadURL* a sensitive information sink [16]). To further investigate this scenario, access to this webpage is required. Static analysis of these scenarios is non-trivial as any static analysis of JavaScript code is challenging due to its highly dynamic nature [17] and as it additionally requires a careful inspection of the various aspects of the *WebView* class and its bridge mechanism. Therefore, it becomes critical for program analyses to fully understand the behavior of *loadUrl* and *addJavascriptInterface*. The aim of our work is to provide this information by performing a large scale study of real-world apps.

IV. Methodology

Figure 1 presents the workflow of LUDroid's analysis framework. It consists of the following modules: *IFCAnalyzer*, *ResourceExtractor*, *UrlAnalyzer*, and *JSAnalyzer*. LUDroid decompiles an APK[1] using APKTool [18]. The decompiled output contains the app's resources and source code in the Smali [19] format. In the *IFCAnalyzer* module a backward

[1]An APK is the binary output of an Android application.

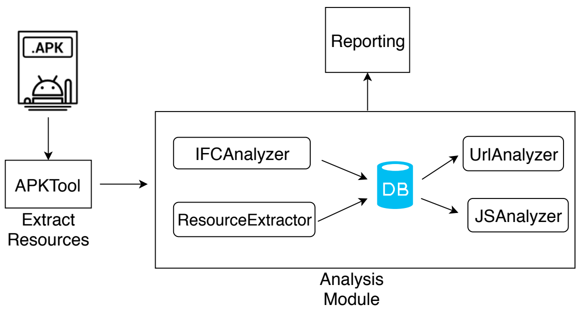

Fig. 1: The workflow of LUDroid

slice (A list of statements that influence a statement [20]) for the method *addJavascriptInterface* is computed to analyze the information flows from the Android to the Javascript side. The *ResourceExtractor* module extracts the resources that are passed as parameters to the method *loadUrl*, like the protocol of the passed URL. This information is stored in a database and passed as an input to the modules *UrlAnalyzer* and *JSAnalyzer*. The *UrlAnalyzer* module analyzes the URLs provided as String argument to the *loadUrl* method. It validates the URLs and extracts various features, like the used protocol, which facilitates the analysis of URLs in hybrid communication. Similarly, the *JSAnalyzer* module analyzes the JavaScript code that is passed to the *loadUrl* method. In the followings we discuss each module in detail.

A. *IFCAnalyzer*

The aim of this module is to understand the nature of the information flow from Android to JavaScript. We label a piece of information as sensitive if leaking it will violate its owner's privacy. To this end, we leverage sensitive sources defined by [16] to identify the sensitive information in Android. In particular we answer the following research questions:

- **RQ1.1:** *How pervasive is information flow from Android to JavaScript?*
- **RQ1.2:** *Do these information flows include sensitive information?*

Figure 2 describes the workflow of the *IFCAnalyzer* module. For every occurrence of the method *addJavascriptInterface* we compute its backward slice to identify the corresponding *WebView* initialization. The *addJavascriptInterface* method injects the Android object into the corresponding *WebView*. It takes two parameters: The first is a Java object, the second is the name of the associated object. If *Javascript* is enabled on this *WebView*, the loaded web pages can invoke the methods exposed by the shared Java object (cf. Listing 2). As we are interested in the exposed functionality of this Java object, we extract the corresponding class and supported methods: Not all methods of the Java object are bridged. Only methods annotated with *@JavascriptInterface* are made available to Javascript. We then identify (potential) sensitive information flows originating from these methods: Javascript could invoke these methods to

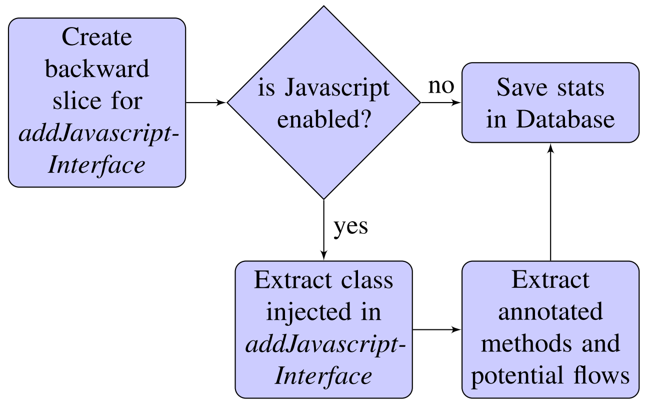
Create backward slice for addJavascript-Interface
is Javascript enabled?
no
Save stats in Database
yes
Extract class injected in addJavascript-Interface
Extract annotated methods and potential flows

Fig. 2: The workflow of IFCAnalyzer

leak the returned sensitive information. Finally, we store the analysis results into a database.

B. *ResourceExtractor*

The *loadUrl* method loads a specified URL given as string parameter (cf. Listing 1, line 11). If the input string starts with *"javascript:"*, the string is executed as JavaScript code (cf. Listing 1, line 9). The aim of this module is to extract the URLs and Javascript code passed to the *loadUrl* method. We create an intra-procedural backward slice on *loadUrl*, extract the input strings, and store them along with their originating class' name. As input strings are often constructed via various String operations, (e.g., using *StringBuilder* to concatenate strings), we extend LUDroid with domain knowledge on the semantics of the Java String class. LUDroid understands the Smali signature of String methods and applies partial evaluation to infer strings created by various String manipulation methods. However, at the moment we do not support complex string operations such as array manipulation. Future work may extend the support for such operations and simple obfuscations. At the moment we concentrate on the features of the *loadURL* parameters that can be extracted with reasonable effort. The output of this module is fed to the *UrlAnalyzer* and *JSAnalyzer* modules to interpret and categorize the URLs and Javascript.

C. *URLAnalyzer*

URLAnalyzer has two functions: (1) it checks the validity of a URL, and (2) extracts its essential features. *URLAnalyzer* reads the passed URL inputs from *ResourceExtractor*, parses them, and extracts the following set of features:

- **Protocol** - The application layer protocol used within the URL, e.g., HTTP.
- **Host** - This can either be a fully qualified domain or an IP address of the corresponding host.
- **Port** - The port (if specified) of the host the request is sent to.
- **Path** - The path (if specified) on the host the request is sent to. Such a path can for example be specified for HTTP or FTP URLs, but also for local file URLs.
- **Search** The search part (if specified) of HTTP URLs. This is the remainder of a HTTP URL after the path, e.g., "?x=5&y=9".
- **Fragment** The fragment is an optional part of the URL that is placed at the end of the URL and separated by a #.

RFC 3986 [21] defines the specification of an URL in augmented Backus-Naur form. *URLAnalyzer* validates each provided URL against this definition in order to detect malformed URLs. For every URL, *URLAnalyzer* either confirms that the URL was correctly built or prints a detailed message why the URL is malformed.

We also categorize the URLs that are created by third-party libraries (SDKs). These libraries use *loadUrl* to load their custom URLs and provide the intended functionality to other app developers, e.g., Facebook SDK for Android provides Facebook authentication service to other apps. Finally, *URLAnalyzer* outputs a database containing the analysis results, that are to be reported by the *Reporting* module.

With respect to the above features we answer the following questions:

- **RQ2.1:** *What is the distribution of protocols used in loadURL?*
- **RQ2.2:** *What percentage of URLs point to files on the device that are assumed to be trusted as they were packed together with the application?*
- **RQ2.3:** *What is the distribution of hosts? Do host hotspots exist, i.e., hosts that requests are being sent to from many different applications?*
- **RQ2.4:** *What is the distribution of resource access within one host discriminated by its path?*
- **RQ2.5:** *What percentage of URLs leverage unencrypted network communication e.g., HTTP, FTP?*
- **RQ2.6:** *Which of the external SDKs cannot be identified and are considered untrusted?*

D. *JSAnalyzer*

JSAnalyzer uses the strings constructed by *ResourceExtractor* and builds a database summarizing the patterns used in raw javascript passed to *loadUrl*. The results in the database are further manually analyzed for the features mentioned in the subsequent paragraphs. The components of JSAnalyzer primarily consist of scripts for automation and reporting.

1) Information Flow from Javascript to Android.: The Android SDK allows app developers to annotate *setter* methods with *JavaScriptInterface*. This supports the reuse of existing web-based functionality in Android by transmitting the results from a web-based/JavaScript method to the Android object. It creates an information flow from the external web application to the Android app. In this paper, we identify use-cases of this behavior.

2) Obfuscated and unsecured Code.: Many third-party library developers use code obfuscation to protect their intellectual property. It is also possible to inject remote third-party libraries in JavaScript using unsecured protocols such as HTTP.

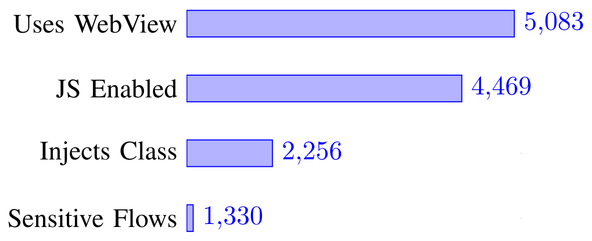

Fig. 3: Hybrid API usage (Over 7500 apps)

TABLE I: Top ten app categories with type of information shared from Android to JavaScript

App category	Type of information
Social	Cookies, File system
Entertainment	Account Information, File system, Network Information, Location
Music & Audio	Account Information, File system, Network Information
LifeStyle	Activity Information, Application level navigation affordances, Locale
Board Games	Date and Time, Location, Network Information
Communication	Activity Information, File system, Location, Network Information
Personalization	Activity Information, Account Information, File system, Location
Books & Reference	File System, Location, Network Information
Puzzle	File System, Internal Memory Information, Location
Productivity	File System, Internal Memory Information, Network Information

In this work we identify the patterns in which these libraries are obfuscated or used insecurely.

3) Pass Native Information to Third Parties.: Many apps pass device specific native information to third-party libraries. This user-sensitive information is leveraged by third-party libraries to enhance their services. However, it can be detrimental to the privacy of the user. In this work, we identify cases of passing sensitive information to third-parties.

In particular, we answer the following questions

- **RQ3.1**: *How frequent is third-party script injection used in raw JavaScript passed to* loadUrl*?*
- **RQ3.2**: *Is there non-trivial information flow from JavaScript to Android?*
- **RQ3.3**: *Do third-party libraries that use* loadUrl *leverage obfuscation?*

V. Evaluation

We evaluated LUDroid on 7,500 random applications from the Google Play Store (published between 2015-2018) to understand hybrid apps' communication patterns in the wild. In the following we answer the research questions raised in the aforementioned analysis modules.

All experiments were performed on a MacBook Pro with a 2,9 GHz Intel Core i7 processor, 16 GB DDR3 RAM, and MacOS Mojave 10.14.1 installed. We used a JVM version 1.8 with 4 GB maximum heap size.

A. IFC from Android to Javascript

RQ1.1: *How pervasive is information flow from Android to JavaScript?* Figure 3 provides the distribution of apps based on various characteristics of hybrid communication. 68% out of 7,500 apps use WebView at least once, i.e. are hybrid apps, which is a significantly high percentage. As JavaScript is not enabled by default, 87.9% of hybrid apps enable JavaScript while the remaining 12.1% use WebView solely for static webpages. Half of the components enabling JavaScript establish an interface to JavaScript via *addJavascrtiptInterface* and bridge an Android object to JavaScript. Therefore, 30% of the apps used in our dataset and 43% of the hybrid apps transfer information from Android to JavaScript. Table I presents the top ten app categories and the corresponding types of information shared with JavaScript. Note that in this work we do not investigate what happens to this data on the JavaScript side, i.e., whether it actually leaks to some untrusted entity. The focus instead is to identify scenarios in the wild that need to be taken into consideration when attempting to design an analysis for hybrid apps.

TABLE II: App components with the shared sensitive information (from Android to JavaScript)

App Name	Category	Component Name	Information shared
Instagram	Social	BrowserLiteFragment	Cookies
TASKA AR MARYAM	Entertainment	Map26330	Location (GPS)
Classical Radio	Musik & Audio	MraidView	External storage file system access, Network Information
BLive	Lifestyle	LegalTermsNewFragment	Location
Cat Dog Toe	Board Games	appbrain.a.be	Location, Network Information
N.s.t. A-Tech	Communication	ax	Location, Network Information
Pirate ship GO Keyboard	Personalization	BannerAd	Device ID, Device's Account information, Locale
IQRA QURAN	Books & Reference	Map26330	Location
Logic Traces	Puzzle	SupersonicWebView	Location
FLIR Tools Mobile	Productivity	LoginWebActivity	Network Information

RQ1.2: *Do these information flows include sensitive information?* 18% of the total apps in our dataset share sensitive information from Android to JavaScript. LUDroid finds 6375 sensitive information flows from Android to JavaScript: Only 18% of these flow to URLs located inside the app, i.e., using the *file* protocol. Note that the inclusion of JavaScript code into an app does not guarantee its trustworthiness, as third party code is regularly included into apps. Thus 82% (or more) of the sensitive information flows could leak to potentially untrusted code. Table II presents 10 randomly selected apps[2] for each category mentioned in Table I along with their corresponding components and shared sensitive information. The majority of these flows include location information, network information and file system access. Starting from HTML5, various web APIs provide access to sensitive information such as the geographical location of a user. In contrast to Android's permission system where users need to approve the permissions just once (potentially in a completely different context), web users would need to approve the access each time or they can provide it for one day. It appears that this might be one of the reason that developers rather propagate sensitive information from Android to the Web, but this compromises users' privacy.

B. URL statistics

LUDroid resolves 3075 distinct URLs. In addition it finds 4980 URLs dynamically created using SDKs. Figure 4 shows the distribution of protocols in the resolved URLs passed to the *loadURL* method (**RQ2.1**). 40.81% of the URLs use the trusted *file* protocol pointing to the device's local files, while the remaining point to external (potentially trusted) hosts (**RQ2.2**). Naturally, developers have more control over these offline local files. While this is good for trusted entities, malicious

[2]Due to the size limitation we could not publish the entire list.

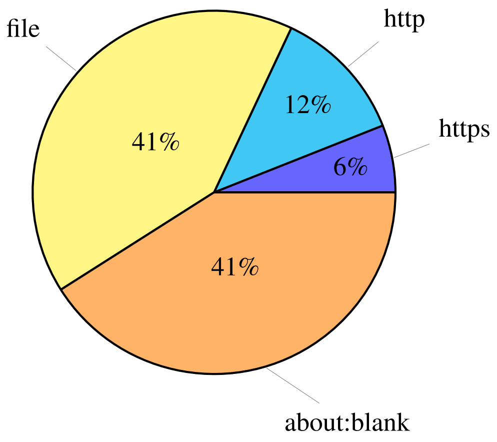

Fig. 4: Distribution of protocols (rounded values for clarity)

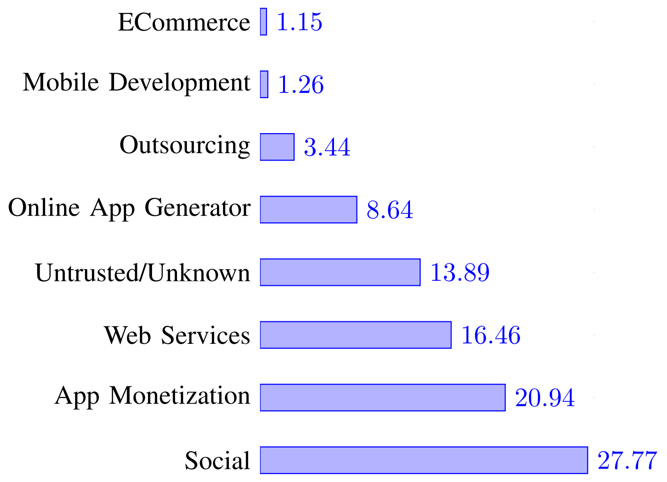

Fig. 5: Distribution of SDK usage in apps by categories (Top eight)

Listing 3: WebViewActivity class in com.zipperlockscreenyellow (manually translated to Java and simplified)

```java
webView.removeAllViews();
webView.clearHistory();
webView.clearCache();
webView.loadURL("about:blank");
```

entities could easily launch phishing attacks by designing offline pages that look similar to trusted web pages. Only good user practices can prevent these attacks from happening: Ideally, APKs should not be downloaded from other sources than the official Play Store. Additionally, users should properly verify app metadata and permissions. As local web pages come bundled with the APK files, they can be taken into account during analysis. However, an analysis might need to consider several security aspects such as, identifying phishing attacks, discovering privacy leaks, or finding keyloggers.

In addition to local file URLs, we discovered that in 41.24% of the resolved cases the URL argument was *"about:blank"*, which displays an empty page. According to Android's WebView [5] documentation *about:blank* should be used to "reliably reset the view state and release page resources". As an example, we discovered *about:blank* in the WebViewActivity class of the app *com.zipperlockscreenyellow*. In this class the method *killWebView* releases the view's resources (see Listing 3). After clearing the history and the cache this method opens a blank page in the WebView.

Considering the network URLs, the most-frequently loaded hosts per category in the analyzed apps are listed in Table III. We find that Facebook and Google SDKs are widely used in apps, primarily for authentication purposes. In addition app monetization and customer analytics SDKs are found in 18.04% of the apps (cf. Figure 5). Figure 5 displays all host categories sorted by their share **(RQ2.3)**. We find that a majority of the analyzed apps use social networking SDKs or app monetization SDKs.

Mobile application development frameworks such as Cordova or PhoneGap allow developers to use HTML/CSS and JavaScript to develop Mobile apps. These libraries primarily use bridge communication between native Android and web technologies [22]. In our study we find that 1.26% of the apps use these frameworks for mobile application development (referred in Figure 5 as Mobile Development).

The 535 URLs that point to a network resource only reference 147 distinct paths. This indicates that in many cases identical host and path combinations were requested by several apps **(RQ2.4)**. In approximately 1.68% of the external URLs the host's port was specified. Additionally, 20.37% of the URLs (HTTP/HTTPS) specify an argument pattern.

While evaluating URLs we gained several relevant security insights. We found that 11.87% of the calls of *loadUrl* resulted in unencrypted network traffic, making a total number of 365 communications. Table IV shows five examples of unencrypted HTTP URLs together with the packages in the corresponding app **(RQ2.5)**. The usage of unencrypted protocols with *loadURL* may result in eavesdropping and phishing vulnerabilities. We demonstrate how to exploit such vulnerability in section V-C. Another security threat is caused by untrusted SDKs using *loadURL*. We find a total of 13.89% apps use untrusted SDKs **(RQ2.6)**. However, in this context untrusted may or may not refer to a malicious SDK. It is non-trivial to classify untrusted SDKs as malicious due to

TABLE III: Selected list of Top-8 SDK hosts, its app share, and a common use case found in the top SDK categories

Category	**Host**	**Percent**	**Common Use Case**
Social Networking	Facebook	20.32	Authentication
App monetization	Vungle	1.75	Monetize Apps by targeted advertising
Web Services	Google	10.58	Authentication
Online App Generator	SeattleClouds	5.99	Unknown (obfuscated)
Outsourcing	biznessapps	1.89	Unknown (obfuscated)
Mobile Development	PhoneGap	1.52	Platform-independent development
E-Commerce	Amazon	1.1	Sales
Others	Ons	0.8	Rendering ebooks

TABLE IV: Five out of 365 loadURL calls using the insecure HTTP protocol

Package Name	URL
com.JLWebSale20_11	http://www.dhcomms.com/applications/dh/cps/google/main_agreepage01.html
net.pinterac.leapersheep.main	http://pinterac.net/dev/leapersheep/index.php?viewall=1
com.quietgrowth.qgdroid	http://docs.google.com/gview?embedded=true&url=http://www.rblbank.com/pdfs/CreditCard/fun-card-offer-terms.pdf
net.lokanta.restoran.arsivtrkmutfagi	http://images.yemeksepetim.com/App_Themes/static-pages/terms-of-use/mastercard/mobile.htm
com.cosway.taiwan02	http://ecosway.himobi.tw

absence of common patterns in these SDKs. Therefore, we take a conservative approach where an SDK is untrusted if there is no public information available on the web. Clearly, security testing of untrusted SDKs is imperative to ensure the integrity of one's code. However, many programmers include desired functionality into their projects without considering the security implications.

Another interesting observation is the usage of online app development platforms. These development platforms allow users to build an application with minimal technical effort and programming background. From the collected data, we find using manual inspection that approximately 8.64% (cf. Figure 5) of the apps use an online app generation platform. A potential threat to these applications is that the developer/app provider using the online app generators neither has the knowledge about the internal details of these apps nor do they perform rigorous testing. A recent study on these online app generators (OAG) found serious vulnerabilities in various OAG providers [23]. Again, programmers should not rely blindly in the quality of external tools and perform additional validation of the resulting app's security properties. Unfortunately, OAGs are particularly intriguing to developers with low technical expertise, so the creators of these platforms have a responsibility.

We discover 18 instances (see e.g. URL for *com.quietgrowth.qgdroid* in Table IV) where a call to *loadURL* was used to display a PDF via Google Docs, which is considered a misuse of WebView. To deliver content such as files to users the WebView documentation recommends to invoke a browser through an *Intent* instead of using a WebView [24]. It appears that developers prefer users to stay in the app for viewing the documentation and thus rather use *WebView* to accomplish this task.

C. Vulnerability Case Study: Unprotected URLs

As described in section V-B, one piece of data *URLAnalyzer* determines is whether a URL passed to *loadURL* is unprotected. An unprotected URL is a URL that points to a network resource and is not protected by any cryptographic means (e.g. TLS). In our evaluation we discovered 365 calls to *loadURL* with unprotected URLs, all of which connect via HTTP.

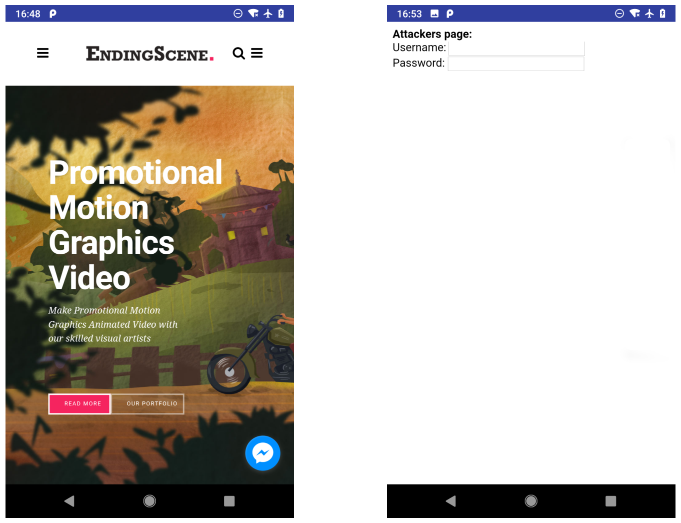
16:48
EndingScene.
Promotional Motion Graphics Video
Make Promotional Motion Graphics Animated Video with our skilled visual artists
READ MORE
OUR PORTFOLIO
16:53
Attackers page:
Username:
Password:

Fig. 6: The EndingScene app without and with being attacked (In a realistic attack the phishing web page may be easily copied from the original web page)

The *loadURL* method embeds a web page into the Android application. When using unprotected URL for *loadURL* an attacker can read the requested webpage, and even more severe, manipulate the server's response that is to be displayed to the user. This is particularly critical as an attacker-controlled webpage is then being shown in the context of a trusted application. The user may be oblivious to the difference between content displayed in a WebView and content displayed in other UI components as WebViews are designed to seamlessly integrate into the native UI components. Depending on the concrete vulnerable application and the placement of the vulnerable WebView in the native UI, various attack scenarios are possible. One of these attack scenarios is a phishing attack where a malicious login page is displayed to the user within the app. As the app is trusted by its users, they are likely to enter their credentials into the phishing page.

Case Study: EndingScene app: To demonstrate the described attack, we have randomly chosen one of many vulnerable applications, EndingScene (v 1.2[3]), a video material promotion app. Immediately in the initial activity, this app loads a webpage and displays it to the user. This scenario is ideal for an attacker, as every user will be presented this initial front page, and the activity consists of nothing else but the front page. In addition, it is very plausible to ask for some type of credential on this front page.

We implemented the network attack using *mitmproxy* [25], a HTTP proxy that can save and manipulate inflowing traffic. We developed a small Python script for use in *mitmproxy*. It substitutes the server's response to the front page request with a self-written malicious login page, which sends the entered credentials to an attacker.

Figure 6 depicts the successful exploitation of the EndingScene app when using our proxy. The left-hand side shows

[3]md5: 7516ddd1bc9d056032ac3173e71251b0

Listing 4: Dynamic Script Loading in loadUrl

```
1 javascript: (function() { var
      script=document.createElement('script');
2 script.type='text/javascript';
3 script.src='http://admarvel.s3.amazonaws.com/
      js/admarvel_mraid_v2_complete.js';
4 document.getElementsByTagName('head')
      .item(0).appendChild(script);}) ()
```

the regular front page of EndingScene while the right-hand side displays the phishing page when the network traffic is being attacked. Evidently, a malevolent entity would create a much more convincing phishing page, our page is for illustration purposes only, to make the attack obvious. This type of attack in general is not new, however, related work [26]–[28] has not detected them in the investigated context of hybrid apps, which may lead to novel attack vectors.

As the described vulnerability is caused by the lack of encryption and signing, it may be avoided by using the transport layer security (TLS) versions of the protocols (e.g. HTTPS, FTPS). Additionally, it is recommendable to make use of certificate pinning in order to prevent threats from corrupted certification authorities.

D. *Javascript statistics*

In this section we present initial insights on JavaScript code that LUDroid identified to be passed to *bridge methods i.e. loadUrl() or evaluateJavascript()*. Based on our manual analysis, we classify the JavaScript code into two categories: (1) involving event-driven functionality using the interface *Event*, and (2) modifying the Document Object Model (DOM) without event-driven functionality. We find that 64% of those trigger event-driven functionality while 31% modify the DOM only. We were unable to resolve 5% of the JavaScript strings owing to our current limitations of *IFCAnalyzer*.

1) Frequent JavaScript Code: We identify a set of 73 distinct JavaScript code snippets passed to *bridge methods* in all investigated apps. Given this low number in relation to the total number, it was not surprising to identify that most of these originate from third-party libraries. Interestingly, the codes identified using *evaluateJavascript* are a subset of those found with *loadURL*, therefore, we will restrict ourselves to the discussion of the latter in the sequel. In what follows, we illustrate the four most interesting cases relevant to understand the developers' intentions.

Case Study: Third party script injection in loadUrl: We identify the case of a third-party script injection that occurs in 3 out of 73 frequent code snippets. One example of such a script is mentioned in Listing 4. Similar instances are present in 8.33% **(RQ3.1)** of the analyzed apps. The script loads a third-party JavaScript code by injecting it into the header of the displayed webpage, resulting in a modification of the global state of the page. In this example, the developers used an unsecured protocol (*HTTP*, cf. Line 3, Listing 4). This scenario makes the webpage and thus the whole Android app susceptible to a man-in-the-middle (MITM) attack, where an attacker can

Listing 5: Modifying the bridged Android object named SynchJS

```
1 javascript:window.SynchJS.setValue((function(){
2     try{
3         return JSON.parse(Sponsorpay.MBE
        .SDKInterface.do_getOffer()).uses_tpn;
4     }catch(js_eval_err){
5         return false;
6 }})());
```

Listing 6: Excerpt of source code for SynchJS object in Listing 5. Source: Github [29]

```
1 public class SynchronousJavascriptInterface {
2     // javascript interface name for adding to web view
3     private final String interfaceName = "SynchJS";
4     private CountDownLatch latch; // Countdown latch to wait
              for result
5     private String returnValue; // Return value to wait for
6     public String getJSValue(WebView webView, String
       expression) {
7       latch = new CountDownLatch(1);
8       String code = "javascript:window." + interfaceName +
            ".setValue((function(){try{return " + expression +
            "+\"\";}catch(js_eval_err){return '';}})());";
9       webView.loadUrl(code);
10      try { // Set a 1 second timeout in case there's an error
11        latch.await(1, TimeUnit.SECONDS);
12        return returnValue;
13      } catch [...] return null; }
14    // Receives the value from the javascript.
15    public void setValue(String value) {
16      returnValue = value;
17      try { latch.countDown(); } catch (Exception e) {} }}
```

intercept the connection and replace the script loaded from `script.src` with malicious JavaScript. However, the user trusts the app and is completely oblivious to the script being downloaded, and the fact that it might be replaced and thus violates the integrity of the app. This attack is implemented in analogy to the attack described in Section V-C, where the login page was substituted by a malicious page.

Case Study: Information flow from JavaScript to Android: Contrary to common intuition we identify interesting cases of information flow from JavaScript to Android in 8.7% **(RQ3.2)** of the investigated apps. Listings 5 and 7 show examples of this behavior.

Listing 5 is particularly interesting as it leverages a synchronous communication channel from Android to JavaScript and back. In Listing 5, a method *setValue()* is invoked on a bridged object *SynchJS*. The method *setValue()* is a setter method defined in the class *SynchronousJavascriptInterface* excerpted in Listing 6. Note that the code of Listing 5 is generated in the method *getJSValue* (line 6), where Android executes the parameter expression in the context of the WebView and waits (line 11) for the thread evaluating the JavaScript code to invoke the bridged *setValue* method. Line 3 in Listing 5 reads the field *uses_tpn* of an object deserialized from a third-party library method *Sponsorpay.MBE.SDKInterface.do_getOffer* and passes that value to the setter method in *SynchJS*. When this method

Listing 7: Information Flow from JavaScript to Android

```
1  javascript :(function() {
2    var metaTags=document.getElementsByTagName('meta');
3    var results = [];
4    for (var i = 0; i < metaTags.length; i++) {
5      var property = metaTags[i].getAttribute('property');
6      if (property && property.substring(0, 'al:'.length) ===
       'al:') {
7          var tag = { "property":
       metaTags[i].getAttribute('property') };
8          if (metaTags[i].hasAttribute('content')) {
9              tag['content'] =
       metaTags[i].getAttribute('content');
10         }
11     results.push(tag); }} // if end
12 window.HTMLOUT
       .processJSON(JSON.stringify(results));})()
```

is invoked inside the WebView's thread, the field *returnValue* is changed (line 16 of Listing 6). The implementation then notifies Android's UI thread via a call to *latch.countDown()*, which basically implements a simple semaphore such that the waiting Android thread can continue its execution and return the value retrieved from the WebView (line 12).

Listing 7 writes meta-tags information of a HTML page to an Android object. Line 5-11 construct an array of objects with properties *property* and *content*. This array is then converted to a string in JavaScript Object Notation (JSON) representation (line 12) before being passed to the *processJSON()* method of the bridged object *HTMLOUT*. Note that due to the fact that the *processJSON* method runs in a different thread than the regular Android code [24] the Java Memory Model may not allow the Android code to see any changes performed to the state of the bridged (and other) objects unless some form of synchronization is being used as in Listing 6.

Android WebViews feature an event system that reacts to many different events in the WebView. The Android SDK allows to override the default *WebView Chromeless browser* window and specify their own policies and window behavior through extending a Java interface called *WebViewClient*. Interestingly, we also identified many similar codes during handling of *WebViewClient* events. As an example, developers can modify the behavior when e.g. the WebView client is closed. Our study finds that many developers transfer results from JavaScript to Android after the WebView client terminates by modifying the *onPageFinished()* method in the *WebViewClient* interface to invoke JavaScript.

This case-study shows the use of sophisticated patterns by developers for communication from *Android* to *JavaScript* and vice-versa. Our study shows intricate cases of using setter methods to permit non-trivial dataflow from JavaScript to Android, in some cases even using (required) synchronization.

Restricting the bidirectional communication impacts the flexibility provided by WebView to developers. Instead static analysis techniques could be leveraged to detect and report similar insecure data flows. However, a simple context-insensitive static analysis on Listing 5 using approaches such as HybridDroid [8], or Bae et. al. [30] will be unsound. The

Listing 8: Leaking Sensitive Information

```
1  javascript : function actionClicked(m,p) {
2      var q = prompt('vungle:'+JSON.stringify(
3          {method:m,params:(p?p:null)}));
4      if (q&&typeof q === 'string'){return
           JSON.parse(q).result;}};
5      function noTapHighlight(){
6          var l=document.getElementsByTagName('*');
7          for(var i=0; i<l.length; i++){
8              l[i].style.webkitTapHighlightColor=
                                       'rgba(0,0,0,0)';
9      }};
10     noTapHighlight();
11     if (typeof vungleInit == 'function') {
12         vungleInit($webviewConfig$);}}
```

Listing 9: Complex control flow via JavaScript

```
1  javascript :(function() { Appnext.Layout.destroy('internal');
       })()
```

unsoundness stems from the analyses' limitation to analyze the described callback communication methods, thus only supporting one-way communication from Android to JavaScript. A precise and sound static analysis would need to consider these non-trivial methods of callback communication that establish a two-way communication channel.

Case Study: Device Information to Third-party: This case study presents the leak of device information to third-party libraries. Listing 8 is taken from an advertising library *Vungle*. Line 8 removes the highlight color from each element. Therefore, the function *noTapHighlight()* makes it susceptible to a confused-deputy attack such as clickjacking. It obscures user clicks, letting the user click on the advertisement without their knowledge. Additionally, Line 11 can potentially leak Vungle's *WebView* configuration object that identifies a device to some web server, in this example through the function *vungleInit()*. *WebView* settings have sensitive information about the host devices that is also used by *WebViewClient*.

Case Study: Code obfuscation in Third-Party libraries: This case study shows an interesting obfuscation pattern using *loadUrl* to deliberately prevent program analyzers from inferring the intended functionality. Need for obfuscation arises from concerns about keeping the intellectual property, or from trying to hide debatable or, worse, malicious behavior.

Appnext is an ad-library which is widely used for app monetization. Listing 9 shows a code snippet found in its library code. In this code a Java object *Appnext* is being bridged and used in JavaScript invoked from Android. This functionality could have been directly implemented in Android/Java itself. It is unclear why the programmers chose to implement it by crossing a language-bridge from Android to JavaScript and back, which is similarly expensive as an *eval* in JavaScript instead of the direct invocation. We have discovered this pattern in 25% **(RQ3.3)** of the apps, which makes this potential obfuscation pattern prevalent among Android apps.

By introducing another layer of complexity added to inter-language analysis, obfuscation increases the difficulty for program analysis tools to infer the actual functionality. A precise and sound analysis of these patterns is required for useful analysis results. Obfuscation patterns in Android apps are discussed in detailed in a recent large scale study [31].

VI. Discussion

Using LUDroid we were able to derive a plenitude of novel statistics covering information flow information, URL statistics and statistics on JavaScript code. Additionally, we detected URL based vulnerabilities. At this point LUDroid is not a stand-alone analysis tool but merely supports manual inspection and calculation of statistical data. The goal of this work is to present interesting insights on how the bridge between Android and JavaScript is used in the wild in order to facilitate the design of automatic program analysis that take both sides of the hybrid app into account. We will share our collected data after acceptance of this manuscript.

Other common limitations of static analysis are native code, reflection, dynamic control flow and obfuscation and the fact that strings like the URLs passed to *loadURL* may be constructed at runtime. All of these obstacles have been investigated in separate lines of research [32]–[37] so we consider them orthogonal to the insights we are aiming at in this study. Note however, that *loadURL* is also a dynamic language feature that—like reflection—may execute code that is constructed at runtime based on a string parameter. Insights gained in studies that target dynamic code execution (e.g. for JavaScript [38]) are also relevant to understand the semantics of *loadURL*.

VII. Related Work

Rizzo et al. [11] proposed BabelView, which models Javascript as a blackbox. They leverage static taint analysis to detect unwanted information flows and five different vulnerability types. Zhang et al. [39] performed a large scale study of the WebView APIs to classify them into four categories of web resource manipulation. Hidhaya et al. [12] described the "supplementary event-listener injection attack" in Android WebViews. They further proposed a tool for automated detection of this vulnerability and a mitigation. Li et al. [40] discovered a new type of WebView-based attack that they call Cross-App WebView Infection (XAWI). Mandal et al. [13] proposed a static analysis tool to detect various vulnerabilities in Android Infotainment applications. Their approach is based on Julia, a static abstract interpretation analysis tool. Fratantonio et al. [14] proposed a static analysis tool to detect malicious application logic in Android apps. Their approach is based on various known static analysis techniques such as symbolic execution and inter-procedural control-dependency analysis. In contrast to these approaches, our work is not limited to specific vulnerabilities, but provides useful insights by inspecting both Android and Javascript code.

Lee et al. [8] proposed HybriDroid, an information flow analysis tool based on WALA. They discussed the semantics of WebView communication including type conversion semantics between Java and Javascript. In contrast to our work, Lee et al. do not analyze the behavior of *loadURL*. In addition, their approach is restricted to the taint analysis of the Information flow from Android to Javascript, and thus miss other insights.

Neugschwandtner et al. [41] proposed two attack scenarios based on when the client or server is compromised. Their approximation is quite coarse in case of privacy leakage where a trusted channel could leak more than the required information. Mutchler et al. [42] conducted a large-scale study of apps using WebView aiming at the security vulnerabilities present in these apps. However, this study focuses only on particular types of vulnerabilities and they did not consider the misuse of Javascript in `loadURL`. Yang et al. [15] examined so called "Origin Stripping Vulnerabilities" caused by wrongly using the *loadURL* method. Bae [30] formalized the semantics of the android interoperations between Java and Javascript. Their approach proposed a type-system based error detection for `MethodNotFound` errors. However, their approach does not consider the information flow from JavaScript to Android Java. In addition to a large scale study, Kim et al. [43] leveraged abstract interpretation to design a static analysis that finds privacy leaks in android applications. Targeting excess authorization and file-based cross site scripting attacks, Chin et al. [44] proposed Bifocals, a tool to detect these vulnerabilities. However, these analyses focused on one particular part of the problem. Our study is targeted at analyzing all programming patterns which may potentially lead to vulnerabilities.

In general the analysis of unencrypted communication in Android apps is a well-explored topic [26]–[28]. For example, Pokharel et al. [26] demonstrated eavesdropping attacks on VoIP apps. However, to the best of our knowledge no previous work has analyzed the security consequences of unencrypted communication caused by *loadURL*.

VIII. Conclusion

In this work, we present a large-scale analysis of *loadURL* usages in real world applications. The statistical results include numerous features, such as information flow data, URL statistics and javascript code features on a set of 7,500 randomly selected applications from the Google Playstore. We implemented our semi-automatic analysis approach in a tool called LUDroid that computes the data by using slicing techniques. As a side effect LUDroid discovered many instances of a vulnerability considering the usage of unprotected protocols in URLs. To demonstrate the validity of these vulnerabilities we exemplarily showcased the exploitation of one of them. The insights gained in this study provide valuable input for designing program analysis that are to analyze hybrid Android apps.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project SmartPriv (16KIS0760).

References

[1] GS, "Android global market share," http://gs.statcounter.com/os-market-share/mobile/worldwide, April 2018.

[2] S. Insight, "Statistics on consumer mobile usage," May 2018. [Online]. Available: https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

[3] D. a. F. Chris Klotzbach, M. Lali Kesiraju, and A. M. at Flurry, "Flurry state of mobile 2017," January 2018. [Online]. Available: http://flurrymobile.tumblr.com/post/169545749110/state-of-mobile-2017-mobile-stagnates

[4] M. Rosoff, "Facebook is offcially a mobile-first company," https://www.businessinsider.in/Facebook-is-officially-a-mobile-first-company/articleshow/49680725.cms, November 2015.

[5] Google, "Webview," September 2018. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView

[6] ——, "Webview for android," December 2018. [Online]. Available: https://developer.chrome.com/multidevice/webview/overview

[7] Ionic, "Developer survey," 2018. [Online]. Available: https://ionicframework.com/survey/2017#trends

[8] S. Lee, J. Dolby, and S. Ryu, "Hybridroid: static analysis framework for android hybrid applications," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 250–261.

[9] 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*, ser. CCS '14, New York, NY, USA, 2014.

[10] M. Shehab and A. AlJarrah, "Reducing attack surface on cordova-based hybrid mobile apps," in *Proceedings of the 2Nd International Workshop on Mobile Development Lifecycle*. New York, NY, USA: ACM, 2014.

[11] C. Rizzo, L. Cavallaro, and J. Kinder, "Babelview: Evaluating the impact of code injection attacks in mobile webviews," *arXiv preprint arXiv:1709.05690*, 2017.

[12] S. F. Hidhaya, A. Geetha, B. N. Kumar, L. V. Sravanth, and A. Habeeb, "Supplementary event-listener injection attack in smart phones," *KSII Transactions on Internet and Information Systems (TIIS)*, vol. 9, no. 10, pp. 4191–4203, 2015.

[13] A. K. Mandal, A. Cortesi, P. Ferrara, F. Panarotto, and F. Spoto, "Vulnerability analysis of android auto infotainment apps," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*. ACM, 2018, pp. 183–190.

[14] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 377–396.

[15] G. Yang, J. Huang, G. Gu, and A. Mendoza, "Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 742–755.

[16] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *NDSS*, 2014.

[17] K. Sun and S. Ryu, "Analysis of javascript programs: Challenges and research trends," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, p. 59, 2017.

[18] C. T. Ryszard Wiśniewski, "Apktool," https://ibotpeaches.github.io/Apktool/.

[19] B. Gruver, "Smali/baksmali." [Online]. Available: https://github.com/JesusFreke/smali

[20] M. Weiser, "Program slicing," *IEEE*, vol. 10, no. 4, pp. 352–357, Jul. 1984.

[21] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (uri): Generic syntax," W3C, Tech. Rep., 2004.

[22] A. PhoneGap, "Phonegap native bridge," aug 2010. [Online]. Available: "https://phonegap.com/blog/2010/08/13/phonegap-native-bridge/"

[23] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes, "The rise of the citizen developer: Assessing the security impact of online app generators," in *2018 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2018, pp. 102–115. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2018.00005

[24] Google, "Building web apps in webview," dec 2018. [Online]. Available: https://developer.android.com/guide/webapps/webview

[25] A. Cortesi and M. Hils, "mitmproxy," https://mitmproxy.org.

[26] S. Pokharel, K.-K. R. Choo, and J. Liu, "Can android voip voice conversations be decoded? i can eavesdrop on your android voip communication," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 7, p. e3845, 2017.

[27] D. He, M. Naveed, C. A. Gunter, and K. Nahrstedt, "Security concerns in android mhealth apps," in *AMIA Annual Symposium Proceedings*, vol. 2014. American Medical Informatics Association, 2014, p. 645.

[28] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 50–61.

[29] G. Genovese, "Github," aug 2012. [Online]. Available: "https://github.com/gcesarmza/androidSamples/blob/master/webview/src/main/java/com/gustavogenovese/webview/SynchronousJavascriptInterface.java"

[30] S. Bae, S. Lee, and S. Ryu, "Towards understanding and reasoning about android interoperations," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, 2019. [Online]. Available: https://plrg.kaist.ac.kr/lib/exe/fetch.php?media=research:publications:icse19.pdf

[31] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 222–235. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274726

[32] S. Groß, A. Tiwari, and C. Hammer, "Pianalyzer: A precise approach for pendingintent vulnerability analysis," in *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. López, J. Zhou, and M. Soriano, Eds., vol. 11099. Springer, 2018, pp. 41–59. [Online]. Available: https://doi.org/10.1007/978-3-319-98989-1_3

[33] J. López, J. Zhou, and M. Soriano, Eds., *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 11099. Springer, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-98989-1

[34] A. Tiwari, S. Groß, and C. Hammer, "IIFA: modular inter-app intent information flow analysis of android applications," *CoRR*, vol. abs/1812.05380, 2018. [Online]. Available: http://arxiv.org/abs/1812.05380

[35] Z. Kan, H. Wang, L. Wu, Y. Guo, and G. Xu, "Deobfuscating android native binary code," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 322–323. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00135

[36] L. Li, T. F. Bissyand'e, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931044

[37] N. Grech, G. Fourtounis, A. Francalanza, and Y. Smaragdakis, "Heaps don't lie: Countering unsoundness with heap snapshots," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 68:1–68:27, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133892

[38] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *ECOOP 2011 – Object-Oriented Programming*, M. Mezini, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52–78.

[39] X. Zhang, Y. Zhang, Q. Mo, H. Xia, Z. Yang, M. Yang, X. Wang, L. Lu, and H. Duan, "An empirical study of web resource manipulation in real-world mobile applications," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*. USENIX Association, 2018, pp. 1183–1198.

[40] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 829–844.

[41] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A view to a kill: Webview exploitation," in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/leet13/workshop-program/presentation/Neugschwandtner

[42] P. Mutchler, J. Mitchell, C. Kruegel, and G. Vigna, "A large-scale study of mobile web app security," 2015. [Online]. Available: http://www.ieee-security.org/TC/SPW2015/MoST/papers/s2p3.pdf

[43] J. Kim, Y. Yoon, K. Yi, and J. Shin, “ScanDal: Static analyzer for detecting privacy leaks in android applications,” in *MoST 2012: Mobile Security Technologies 2012*, H. Chen, L. Koved, and D. S. Wallach, Eds. Los Alamitos, CA, USA: IEEE, May 2012. [Online]. Available: http://ropas.snu.ac.kr/scandal/

[44] E. Chin and D. Wagner, “Bifocals: Analyzing webview vulnerabilities in android applications,” in *Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267*, ser. WISA 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 138–159. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05149-9_9