The Journal of Systems & Software 170 (2020) 110775

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A Large Scale Analysis of Android — Web Hybridization ☆

Abhishek Tiwari [a,1], Jyoti Prakash [b,*], Sascha Groß [b], Christian Hammer [b]

[a] National University of Singapore, Singapore City, Singapore
[b] University of Potsdam, Potsdam, Germany

ARTICLE INFO

Article history:
Received 2 December 2019
Received in revised form 3 July 2020
Accepted 3 August 2020
Available online 18 August 2020

Keywords:
Android Hybrid Apps
Static Analysis
Information Flow Control

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 a 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 and the 196 most popular applications from the Google Playstore as well as 1000 malware samples. Our results advance the general understanding of hybrid applications, as well as implications for potential program analyses, and the current security situation: We discovered thousands of flows of sensitive data from Android to JavaScript, the vast majority of which could flow to potentially untrustworthy code. Our analysis identified numerous web pages embedding vulnerabilities, which we exemplarily exploited. Additionally, we discovered a multitude of applications in which potentially untrusted JavaScript code may interfere with (trusted) Android objects, both in benign and malign applications.

1. 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 GS, 2018). Various reports (S. Insight, 2018; Chris Klotzbach et al., 2018) reveal that 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 (Rosoff, 2015). 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* (Google, 2018b) component, a chromeless browser (Google, 2018c) capable of displaying webpages, embeds the web applications into a view of the Android app. Ionic's developer survey (Ionic, 2018) shows the increasing prevalence of hybrid applications. In previous 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.

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 7500 random applications from the Google Play Store, we found 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 leaking various pieces of sensitive information, such as the user's location.

☆ This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through the project SmartPriv (16KIS0760) and the German Research Foundation (DFG) via the collaborative research center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223), project B02.

* Corresponding author.
E-mail addresses: tiwari@comp.nus.edu.sg (A. Tiwari), jyoti@uni-potsdam.de (J. Prakash), gross@uni-potsdam.de (S. Groß), hammer@cs.uni-potsdam.de (C. Hammer).

[1] While performing this work the author was at the University of Potsdam.

https://doi.org/10.1016/j.jss.2020.110775
0164-1212/

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. Lee et al. (2016) provide a framework for hybrid communication's type error discovery and taint analysis of information flows between Android and JavaScript. However, their framework focuses on discovering type errors. They evaluate their taint analysis on 48 apps only without providing insights into the nature of information flows from Android to JavaScript and vice versa. Besides, they overlook the (mis)usage of URLs and JavaScript inside hybridization APIs. Other related works only consider a very specific vulnerability arising from hybridization (Jin et al., 2014; Shehab and AlJarrah, 2014; Rizzo et al., 2017; Hidhaya et al., 2015; Mandal et al., 2018; Fratantonio et al., 2016; Yang et al., 2018)

Hybrid communication leverages APIs like the *loadUrl* or *evaluateJavascript* 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 performed a study, LUDroid, using 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 discovered 6375 sensitive flows from Android to JavaScript.

The major parameter passed to *loadUrl* is the URL to be loaded. We analyzed the syntax and semantics of each URL and provide a detailed categorization. As a byproduct, we found several vulnerabilities concerning the usage of these URLs. We successfully exploited some of these to demonstrate the threats.

Alternatively, *loadUrl* and *evaluateJavascript* accept raw JavaScript code as a 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 many apps save the runtime state of WebView to Android before destroying that component. These apps perform various privacy leaking functions. These functions include: (1) fingerprinting the device for advertisement or monetization purposes and (2) obfuscating code to preclude analysis of its 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.

Additional studies with the most popular 196 apps from the Google Play Store and 1000 malware samples from the AMD dataset (Wei et al., 2017) show differing properties than those by the general benign apps. The malware ignores *evaluateJavascript* but – even though many are repackaged benign apps – leverage many SDKs, potentially for financial gain. Clickjacking attacks have a similar motivation. Other scenarios involve information theft or injection attacks. In contrast the most popular apps display a peculiar usage of *evaluateJavascript*, which differs from other apps, e.g., to control the page navigation or to inject secrets into web forms (however, these secrets are stored in plain text in the apk). We also found that security-critical popular apps like banking do not use *loadURL*. Many popular apps leverage SDKs from social networks or mobile payment.

Technically we provide the following contribution:

- *Information flow analysis* We thoroughly investigate around 8700 real-world Android apps, both benign and malign. 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 much smaller set distinct code snippets, most of which originating from third-part libraries. We compare the behavior of regular benign apps to the most popular apps and malware and find several significant differences and security flaws. We extract relevant features and highlight their implications on program analysis of hybrid apps.

This paper is an extension of a conference paper (Tiwari et al., 2019). This extended version adds the following main contributions:

- Support for *evaluateJavaScript* (Section 8.2)
- Extension of the evaluation dataset to include an
  - Analysis of the top 196 apps from the top 11 categories in Google Play Store (Section 8.1)
  - Analysis of 1000 samples from 71 families of malware from the Argus AMD malware dataset (Section 9)
- Comparison of the data usage between malware and benign apps (Section 9)
- Brief explanation of the methodology (Section 4.1).

To summarize, our work advances the *state-of-the-art* in understanding the usage in Android-Web hybridization and elucidates their relevant implications on program analysis, particularly for security and/or privacy scenarios. However, the aim of this work is not to discover specific vulnerabilities in Android-Web hybridization, but to provide a set of use cases to validate future implementations of program analyses for hybrid apps.

2. Background

2.1. 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.

2.2. WebView, loadURL, and evaluateJavaScript

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 rather than 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)*. Additional to the URL provided as argument to the first method, the second variant accepts additional headers for the HTTP 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 via the pseudo-protocol *javascript:*. *WebView* leverages the appropriate renderer for each URL type automatically. Finally, a dedicated API *evaluateJavascript* executes JavaScript code directly.

3. 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). Execution of JavaScript in a *WebView* object is disabled by default but can be enabled by overriding its default settings (line 5). A Java interface object can be shared with the WebView to be accessible via JavaScript. Via this object the capabilities of the Android world can be bridged to the Web component. This *bridge communication* allows JavaScript to, e.g., access various sensors' data that are usually only accessible from Android. In our example, an object of the class *Leaker* (see Listing 2) is shared (Listing 1, line 8) with the *WebView* object *myWebView*, such that every webpage loaded into *myWebView* can access 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:* pseudo-protocol to the passed code) from Android, and (2) to load a custom URL (line 11) (which may execute JavaScript code specified or loaded in the web page).

Previous work (Lee et al., 2016) take a first step into analyzing the data flows from Android to JavaScript but is only partially sound, and concentrates on potential type errors when passing data between the two worlds. To improve the understanding of 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*, *loadUrl*, and *evaluateJavascript* 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 10 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 12) where it might be leaked 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 this to be an illicit information flow, as they have no information whether the WebView's code actually leaks the shared data (or do not even consider *loadURL* a potential information sink Rasthofer et al., 2014). To further investigate this scenario, access to the executed JavaScript code is required. However, static analysis of JavaScript code is challenging due to its highly dynamic nature (Sun and Ryu, 2017), 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*, *evaluateJavascript* and *addJavascriptInterface*. The aim of our work is to provide this information by performing a large-scale study of real-world apps.

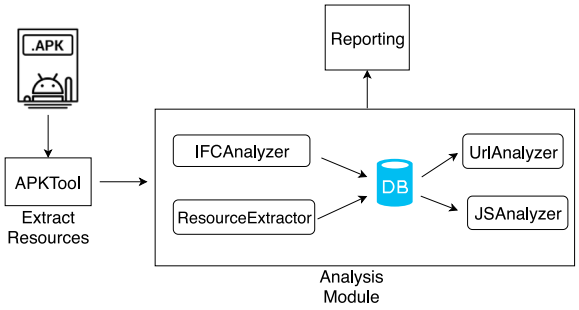

Fig. 1. The workflow of LUDroid.

4. Methodology

We develop the toolchain *LuDroid* to facilitate the semi-automated analysis for discovering the usage of *loadUrl* and *evaluateJavascript*. Fig. 1 presents the workflow of LUDroid's analysis framework, consisting of the following modules: *IFCAnalyzer*, *ResourceExtractor*, *UrlAnalyzer*, and *JSAnalyzer*. LUDroid decompiles an APK[2] using APKTool (Ryszard Wiśniewski, 2020). The decompiled output contains the app's resources as well as source code in the Smali (Gruver, 2020) format . The *IFCAnalyzer* module computes the set of statements that influence the method *addJavascriptInterface* (i.e. the *backward slice* Weiser, 1984). The backward slice is used to analyze which information flows from the Android to the JavaScript side. The *ResourceExtractor* module extracts the strings that are passed as parameters to the two variants of the method *loadUrl* and the method *evaluateJavascript*. This data is stored in a database that is passed as input to the modules *UrlAnalyzer* and *JSAnalyzer*. The *UrlAnalyzer* module analyzes the URLs provided as string argument to the two variants of the *loadUrl* method. It validates the URLs and extracts various features, such as 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* and *evaluateJavascript* methods. In the followings we discuss each module in detail.

4.1. IFCAnalyzer

The aim of this module is to understand the nature of the information flowing from Android to JavaScript. In particular we plan to 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?*

We consider a piece of information to be sensitive if leaking it will violate its owner's privacy. Previous work has identified APIs that return potentially sensitive information (Rasthofer et al., 2014), and we conservatively consider data sensitive if it originates from any of these information sources.

[2] An APK is the binary format of an Android application.

Listing 1: MainActivity.java

```
1  protected void onCreate(Bundle savesInstanceState) {
2  WebView myWebView = (WebView) findViewById(R.id.webview);
3  WebSettings webSettings = myWebView.getSettings();
4 //enable JavaScript on WebView
5  webSettings.setJavaScriptEnabled(true);
6  // add interface object of type Leaker to the WebView's DOM as a property named "Android" of the global object
7   Leaker obj = new Leaker(this);
8  myWebView.addJavascriptInterface(obj, "Android");
9   //case 1: invoke JavaScript from Android
10  myWebView.loadUrl("javascript:"+" print(Android.showToast('Hello World'))");
11  //case 2: load a webpage (potentially executing JavaScript), the object "Android" persists as property of the DOM's global object
12  myWebView.loadUrl( "http://www.dummypage.com");
13 }
```

Listing 2: Leaker.java

```
1 //Add a JavascriptInterface annotation before the method you want to bridge
2 @JavascriptInterface
3 public String showToast(String toast) {
4  TelephonyManager tManager = (TelephonyManager) mContext.getSystemService(Context. TELEPHONY_SERVICE);
5  String uid = tManager.getDeviceId(); // get the device ID
6  return uid;
7 }
```

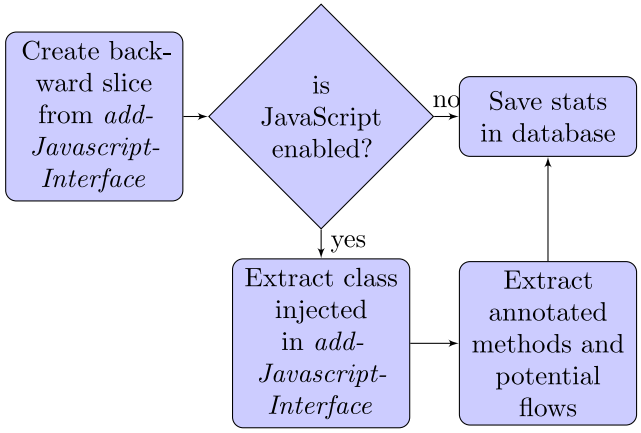

Fig. 2. The workflow of IFCAnalyzer.

Fig. 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 an Android object into the target *WebView*. It takes two parameters, a Java object and the name used to expose this object in this *WebView*'s JavaScript engine. If *JavaScript* is enabled in this *WebView*, the loaded web pages can invoke the methods exposed by the injected Java object (cf. Listing 2). As we are interested in the exposed functionality of this Java object, we extract its class and exposed 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 leak the returned sensitive information. Finally, we store the analysis results into a database.

The backward slices leveraged by *IFCAnalyzer* at the point of this writing are transitively back-tracing explicit information flows (i.e. definition-use chains) for the variables in question. For example, in Listing 1 our analysis computes a backward slice for the parameter *obj* passed to *addJavascriptInterface* and determines it to be of the class *Leaker*. Further, a backward slice for its target object, *myWebView*, returns the object defined in line 2. Based on information computed via backward slicing, *IFCAnalyzer* determines the bridge object, extracts the details of the annotated methods, and stores the results in a database.

Note that information could also be transmitted by implicit information flows (i.e. without def-use chains), however, implicit flows only allow to transmit information bit by bit, e.g. in a loop. As we currently do not support string manipulation via character arrays (cf. Section 4.2), it does not make sense to consider implicit flows when computing backward slices. However, we consider these topics important for a proper analysis that brides the JavaScript and Android worlds in order to preclude certain obfuscations of information flows.

4.2. ResourceExtractor

The *loadUrl* methods load a specified URL given as string parameter (cf. Listing 1, line 12). If the input string starts with *"javascript:"*, the string is executed as JavaScript code (cf. Listing 1, line 10). The aim of this module is to extract the URLs and JavaScript code passed to the *loadUrl* methods and *evaluateJavascript*. We create an intra-procedural backward slice from each of these method calls, extract the string parameters, and store them alongside their originating class' name. As string parameters are often constructed via various *String* operations, (e.g., using the *StringBuilder* class to concatenate strings), we extended LUDroid with domain knowledge on the semantics of the Java *String* class. When LUDroid detects the Smali signature of a *String* or *StringBuilder* method it applies partial evaluation based on the method's semantics to statically infer the result created by the string manipulation. However, at the time of this writing we do not support the automatic resolution of complex string operations based on manipulating elements of a character array. Future work may extend the support for such operations and thus simple obfuscations. For now we concentrate on the features of the string parameters that can be extracted with reasonable effort, in order to gain timely insights on what is required for static analysis of hybrid apps. The output of this module is fed to the *UrlAnalyzer* and *JSAnalyzer* modules to interpret and categorize the URLs and JavaScript.

4.3. URLAnalyzer

URLAnalyzer has two functions: (1) it checks the validity of a URL, and (2) extracts its essential features. *URLAnalyzer* parses the

URLs received from *ResourceExtractor* and extracts the following set of features:

- **Protocol** — The application layer protocol of 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 of the host the request is sent to (optional).
- **Path** — The path on the host the request is sent to (optional). Such a path can for example be specified for HTTP or FTP URLs, but also for local file URLs.
- **Search** The search part of HTTP URLs (optional). 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 #.

RFC 3986 (Berners-Lee et al., 2004) defines the specification of URLs 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 is syntactically correct, or prints a detailed message why the URL is malformed.

We also categorize the URLs created by third-party libraries (SDKs). These libraries use *loadUrl* to provide the intended functionality to app developers, e.g., the Facebook SDK for Android provides Facebook's authentication service. Finally, *URLAnalyzer* creates a database containing the analysis results, so they can 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 bundled 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?*

4.4. JSAnalyzer

JSAnalyzer summarizes patterns found in JavaScript passed to both variants of *loadUrl* and *evaluateJavascript* (i.e. the strings constructed by *ResourceExtractor*). The results are stored in a database for further manual analysis with respect to the features described in the sequel. The components of JSAnalyzer primarily consist of scripts for automation and reporting.

4.4.1. Information flow from JavaScript to Android

The Android SDK permits to annotate *setter* methods with *JavascriptInterface*. Transmitting the results from a web-based/JavaScript method to the Android object supports reuse of existing web-based components in Android. It creates an information flow from the external web application to the Android app. In this paper, we identify use-cases of this behavior.

4.4.2. Obfuscated and unsecured code

Many third-party libraries employ code obfuscation to protect their intellectual property. At the same time it is possible to inject remote third-party libraries in JavaScript using unsecured protocols such as HTTP. In this work we identify patterns in which external libraries are obfuscated or included insecurely.

4.4.3. Passing of sensitive information to third parties

Many apps pass device specific information to third-party libraries. This sensitive information is leveraged by third-party libraries to enhance their services, such as targeted advertising. However, it can be detrimental to user privacy. 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 JavaScript passed to* loadUrl *or* evaluateJavascript*?*
- **RQ3.2**: *Is there non-trivial information flow from JavaScript to Android?*
- **RQ3.3**: *Do third-party libraries leverage obfuscation for their JavaScript code?*

5. Dataset selection

We curate four different datasets containing both benign and malware apps. Our rationale to study both is the following: (1) the analysis of benign apps conveys information on how developers use Android-Web hybridization in practice, and (2) analysis on malware provides relevant insights on the use of the feature for malicious purposes. The former exhibits patterns which can be dangerous and potentially exploited. The latter demonstrates that these exploits have been exploited in practice.

We select the benign apps from the Google Play Store and malware from the Argus AMD dataset (Wei et al., 2017). To obtain the benign apps, we crawled the Google Play Store based on the criteria (A) and (B) below. The Argus AMD dataset is the state of the art malware dataset available to the best of our knowledge. In what follows, we describe the datasets chosen for our study.

A **Benign apps.** In this dataset we randomly chose 7500 apps published on the Google Play Store between 2015–2019.

B **Frequently used apps.** We selected a total of 144 of the top downloaded apps from 11 app categories on the Google Play Store. These categories are Banking, Business, Education, Entertainment, Health, Music, News, Online payments, Shopping, Social, and Travel, and are the categories of apps that were among the highest downloaded on Play Store. In addition, we also selected the top downloaded apps (referred as top apps) across all categories and apps that yield the most revenue (referred as top grossing), consisting of 52 apps. In total, we curate 13 categories of apps. Table 6 lists these categories together with the apps in each category as of Dec 2019, when they were downloaded.

C **Malware:** The Argus AMD dataset contains 24,553 samples from 71 different families of malware. The number of malware in each family range from 4 to 7843. To have a representative from each malware family, we chose at least one from each family, but choosing up to approx. 20% of the malwares from each family. In total, we choose 1000 malware for the study.

6. Evaluation — IFC and URL analysis

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. In the following, we provide the inferences from our evaluation for each of the aforementioned datasets.

We evaluated *LuDroid* on more than 7500 random applications from the Google Play Store to understand hybrid apps' communication patterns in the wild.

Table 1
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

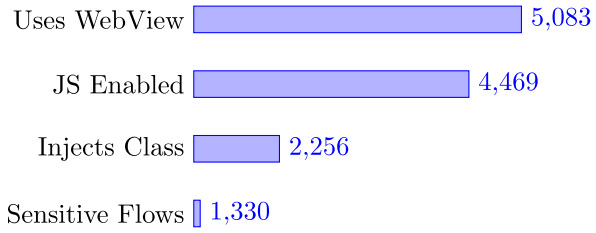

Fig. 3. Hybrid API usage (over 7500 apps).

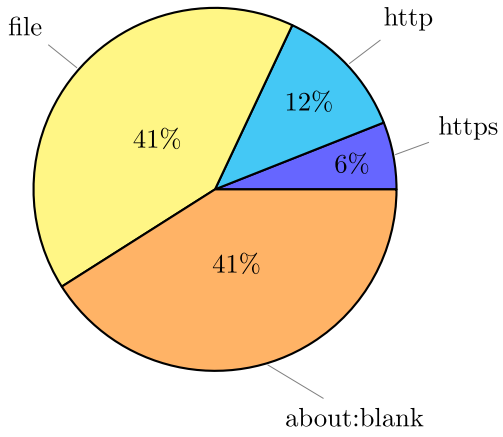

Fig. 4. Distribution of protocols (rounded values for clarity).

6.1. IFC from Android to JavaScript in benign apps

RQ1.1: *How pervasive is information flow from Android to JavaScript?* Fig. 3 provides the distribution of apps based on various characteristics of hybrid communication. 68% out of 7500 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 *addJavascriptInterface* 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 1 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.

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 2 presents 10 randomly selected apps[3] for each category mentioned in Table 1 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 prefer to propagate sensitive information from Android to the Web, at the expense of users' privacy.

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

```
1 webView.removeAllViews();
2 webView.clearHistory();
3 webView.clearCache();
4 webView.loadURL("about:blank");
```

6.2. URL statistics in benign apps

LUDroid resolved 3075 distinct URLs. In addition it found 4980 URLs dynamically created using SDKs. Fig. 4 shows the distribution of protocols in the resolved URLs passed to the *loadURL* method **(RQ2.1)**. 40.81% of the URLs use the *file* protocol pointing to the device's (trusted) 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 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 (Google,

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

Table 2
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

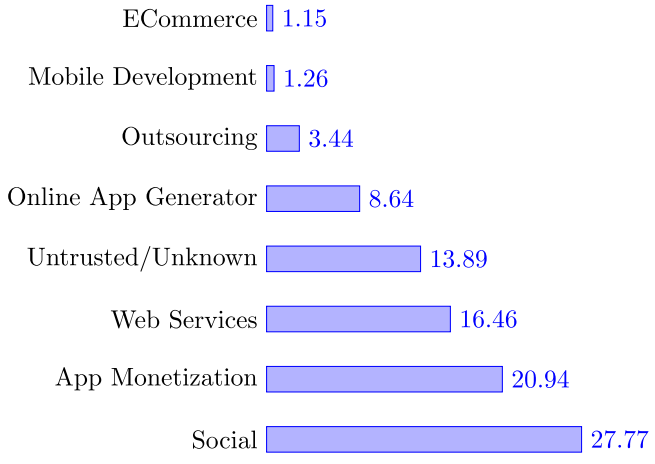

Fig. 5. Distribution of SDK usage in apps by categories (Top eight).

2018b) 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 network URLs, the most-frequently loaded hosts per category in the analyzed apps are listed in Table 3. We found 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. Fig. 5). Fig. 5 displays all host categories sorted by their share **(RQ2.3)**. We found 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 (A. PhoneGap, 2010). In our study we found that 1.26% of the apps use these frameworks for mobile application development (referred in Fig. 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 multiple 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 to *loadUrl* resulted in unencrypted network traffic, making a total number of 365 communications. Table 4 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 a vulnerability in Section 6.3. Another security threat is caused by untrusted SDKs using *loadURL*. We found 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 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 functionalities 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 found using manual inspection that approximately 8.64% (cf. Fig. 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 for various OAG providers (Oltrogge et al., 2018). Again, programmers should not rely blindly on 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 discovered 18 instances (see e.g. URL for *com.quietgrowth.qgdroid* in Table 4) 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 files to users the WebView documentation recommends to invoke a browser through an *Intent* instead of using a WebView (Google, 2018a). It appears that developers prefer users to stay inside the app for viewing documentation, and thus rather use *WebView* to accomplish this task.

6.3. Vulnerability case study: Unprotected URLs

As described in Section 6.2, *URLAnalyzer* determines whether a URL passed to *loadURL* is unprotected, i.e. whether it 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.

The *loadURL* method embeds a web page into the Android application. When using an 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 displayed in the context of a trusted application. The user may be oblivious to the difference between content displayed in

Table 3
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 4
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

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 application and the placement of the vulnerable WebView in the native UI, various attack scenarios are possible. One 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 on the phishing page.

Case study: EndingScene app. To demonstrate the described attack, we randomly chose one vulnerable application, EndingScene (v 1.2[4]), 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 a network attack using *mitmproxy* (Cortesi and Hils, 2020), 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.

Fig. 6 depicts the successful exploitation of the EndingScene app when using our proxy. The left-hand side shows 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 (Pokharel et al., 2017; He et al., 2014; Fahl et al., 2012) has not detected them in the context of hybrid apps, which may lead to novel attack vectors.

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

7. Evaluation — JavaScript in benign apps

In what follows we present initial insights on JavaScript code that LUDroid identified to be passed to *bridge methods*, i.e., *loadUrl* or *evaluateJavaScript*.

[4] md5: 7516ddd1bc9d056032ac3173e71251b0.

7.1. Frequent JavaScript code in benign apps

Based on our manual analysis for benign apps, 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 found that 64% trigger event-driven functionality while 31% modify the DOM only. We were unable to resolve 5% of the JavaScript strings owing to the current limitations of *IFCAnalyzer*. We identified a set of 73 distinct JavaScript code clones (of type 2 Koschke, 2007, i.e. syntactically identical copy where identifiers might have been replaced[5]) 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 set of code executed using *evaluateJavaScript* is a subset of those executed via *loadURL*, therefore, we will restrict ourselves to the discussion of the latter in the sequel.

Table 5 lists the four most frequently used JavaScript code fragments and their usage. The most commonly used JavaScript code snippet in our dataset originates from the Facebook SDK. More than 32% of apps, in our dataset, have used it at least once. This code snippet provides the functionality to authenticate with Facebook on the web in case the Facebook mobile app is not present in the user's device. The second most common JavaScript code snippet exhibits a peculiar case. In this snippet, a DOM element modifies an Android object. We describe further details in Case Study 7.1.2. Similar functionality is manifested by the fourth JavaScript code snippet. In this case, a JavaScript code snippet, originated from an external SDK, modifies an Android object by invoking the setter method. In particular, both cases manifest an interesting scenario where JavaScript modifies Android objects. Finally, the program in the third row originates from the advertisement library Vungle. Details regarding this case are described in Case Study 7.1.3.

In what follows, we illustrate the four most interesting cases relevant to understand the developers' intentions.

7.1.1. Case study: Third-party script injection in loadUrl

We identify the case of a third-party script injection that occurs in 3 of the 73 codes identified. One example of such a

[5] As we only have access to decompiled binaries we are lacking the original identifier names in most cases.

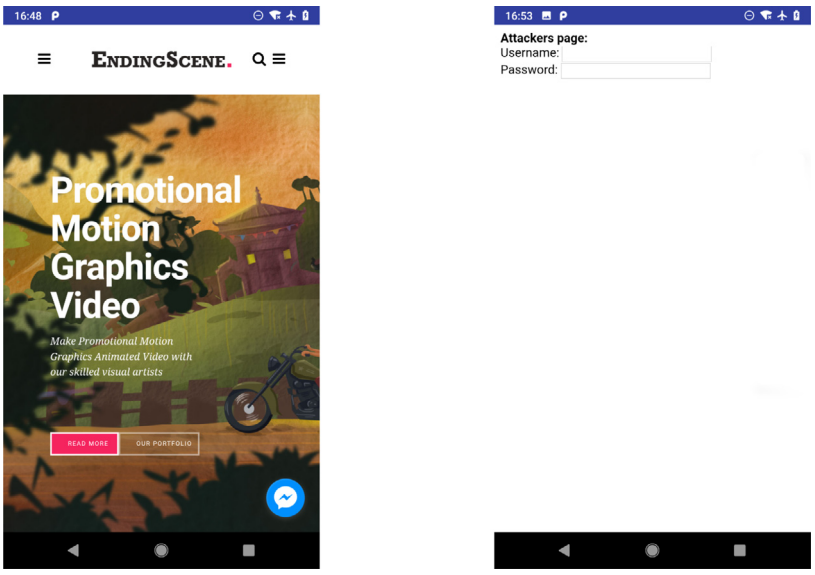

Fig. 6. The EndingScene app without and with being attacked (in a realistic attack the phishing web page may be easily copied from the original web page).

Table 5
Most frequently resolved JavaScript code.

Program fragment	%age of apps	Comments	Randomly selected apps

(function(){var event = document.createEvent('Event');
event. initEvent('fbPlatformDialogMustClose',true,true);
document.dispatchEvent(event);})();
	32.25	Code from Facebook SDK	com.tivion.usa
			com.com014.lotto
			com.t1gamer.lmcs
			com.ape.games.nlu

javascript :window.HTMLOUT. processHTML(document.getElementById('SearchResults'). innerHTML);	16.65	Injecting HTML in bridge object *HTMLOUT*	com.www.Fishingmac com.tc.veda com.alx.roseslwp com.mediaapp.mex com.calcrate

VungleAdjavascript:function actionClicked(m, p) {
var q = prompt('vungle:' + JSON.stringify({ method: m, params: (p ? p : null) }));
if (q && typeof(q) === 'string') {
return JSON.parse(q).result;
}
};
function noTapHighlight() {
var l = document.getElementsByTagName('*');
for (var i = 0; i < l.length; i++) {
l[i].style.webkitTapHighlightColor = 'rgba(0,0,0,0)';
}
};
noTapHighlight();
if (typeof vungleInit == 'function') { vungleInit($webviewConfig$);}
	4.33	Code from Vungle advertising library	Lucky-Wheel Booboo
			Guess-The-Flag
			com.ignm.gesu

javascript:window.SynchJS.setValue((function() { try {return JSON.parse(Sponsorpay.MBE. SDKInterface.do_getOffer()).uses_tpn;} catch(js_eval_err) {return false ;}}) ()) ;	1.9	Bridged Communication with Sponsorpay SDK	com.wTieugiano SocialNetworkCircus BBCNews-Pashto WoRSA

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);})()
```

script is displayed in Listing 4. Similar instances are present in 8.33% **(RQ3.1)** of the analyzed apps. The script loads third-party JavaScript code by injecting a script tag 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 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 that it might be replaced, which violates the integrity of the app. This attack can be implemented analogously to the attack described in Section 6.3, where the login page was substituted by a malicious page.

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

```
1 %% \authornote
2 %% Removed the source from here. Placed it in the text.
3 %% \endauthornote
4 public class SynchronousJavascriptInterface {
5     // JavaScript interface name for adding to web view
6     private final String interfaceName = "SynchJS";
7     private CountDownLatch latch; // Countdown latch to wait for result
8     private String returnValue; // Return value to wait for
9     public String getJSValue(WebView webView, String expression) {
10       latch = new CountDownLatch(1);
11       String code = "javascript:window." + interfaceName +
                ".setValue((function(){try{return " + expression +
                "+\"\";}catch(js_eval_err){return '';}})());";
12       webView.loadUrl(code);
13       try { // Set a 1 second timeout in case there's an error
14         latch.await(1, TimeUnit.SECONDS);
15         return returnValue;
16       } catch [...] return null; }
17     // Receives the value from the JavaScript.
18     public void setValue(String value) {
19       returnValue = value;
20       try { latch.countDown(); } catch (Exception e) {}
21     }
22 }
```

Listing 7: Information Flow from JavaScript to Android

```
1 (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);
12  } // if end
13 } // for end
14 window.HTMLOUT          .processJSON(JSON.stringify(results));
15 }) ()
```

Listing 8: Representative Java listing called from *onPageFinished()* in *Liberty Education App*

```
1 public void process() {
2     ....
3     WebView v0 = p0.viewFinished;
4     ...
5     String v1 = "javascript:..."; //String from Listing 7
6     v0.loadUrl(v1)
7     ...
8 }
```

7.1.2. Case study: Information flow from JavaScript to Android

Contrary to common intuition we identified interesting cases of information flow from JavaScript to Android in 8.7% **(RQ3.2)** of the investigated apps. Listings 5 and 7 display 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 (Source (Genovese, 2012)). Note that the code of Listing 5 is generated in the method *getJSValue* (line 9), where Android executes the parameter expression in the context of the WebView and waits (line 14) 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 is invoked inside the WebView's thread, the field *returnValue* is changed (line 19 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 15).

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 14) 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 (Google, 2018a) 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 developers 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 code fragments during handling of *WebViewClient* events. As an example, developers can modify the behavior when e.g. the WebView client is closed. Our study found 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. Listing 8 shows an example taken from the app *Liberty Education* where the method *process()* is called from method *onPageFinished()* by a series of method calls.

Our 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 (Lee et al., 2016), or Bae et al. (2019) will be unsound. The 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.

7.1.3. Case study: Device information to third-party

This case study illustrates leakage of device information to third-party libraries. Listing 9 is taken from the advertising library *Vungle*. Line 10 removes the highlight color from each

Listing 9: Leaking Sensitive Information

```
1 function actionClicked(m,p) {
2     var q = prompt('vungle:' + JSON.stringify({method:m, params:(p?p:null)}));
3     if (q && typeof(q) === 'string') {
4         return JSON.parse(q).result;
5     }
6 };
7 function noTapHighlight(){
8     var l=document.getElementsByTagName('*');
9     for(var i=0; i<l.length; i++){
10        l[i].stylewebkitTapHighlightColor= 'rgba(0,0,0,0)';
11    }
12 };
13 noTapHighlight();
14 if (typeof vunglelnit == 'function') {
15     vunglelnit($webviewConfig$);
16 }
```

Listing 10: Complex control flow via JavaScript

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

element. Therefore, the function *noTapHighlight()* makes the app susceptible to a confused-deputy attack such as clickjacking. It obscures user clicks, making users click on advertisements without their knowledge. Additionally, Line 14 can potentially leak Vungle's *WebView* configuration object, which contains identifiable information of a device, to some web server, in this example through the function *vungleInit()*. *WebView* settings contain sensitive information about the host device that is also used by *WebViewClient*.

7.1.4. 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 safeguarding 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 10 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 even more 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. In this case, the anonymous function can be directly replaced by the function call.

In our study, we found an instance of obfuscation where the *WebView* class itself is obfuscated. Various libraries inherit from the *WebView* class and define their own subclass to access *WebViews* functionality. These subclasses are then obfuscated using a code-obfuscator tool. Listing 11 shows an instance from the library code *Ad-Locus*. Listing 11 defines a class `AdLocusAdapter` which contains the obfuscated method names such as `a()`,`b()` and `c()`. To improve precision, static analysis needs to consider these libraries and especially the obfuscation patterns present in libraries.

By adding another layer of complexity 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 (Wermke et al., 2018).

Listing 11: Representative Java listing of obfuscation in Library Code-AdLocus

```
1 package com.adlocus.adapters;
2 public class AdLocusAdapter {
3     ...
4     protected final WeakReference a;
5     ...
6     private WebView b;
7     ...
8     // direct methods
9     ...
10    public AdLocusAdapter(AdLocusLayout v) { ... }
11    private static WebView a(AdLocusAdapter v) { ... }
12    private void a(AdLocusLayout v) { ... }
13    ...
14    // virtual methods
15    public void a() { ... }
16    public void b() { ... }
17    public void c() { ... }
18    ...
19 }
```

Listing 12: Code from Facebook SDK found in all of the top apps category

```
(function() { var event = document.createEvent('Event');
    event.initEvent('fbPlatformDialogMustClose', true,true);
    document.dispatchEvent(event);
})();
```

8. Evaluation — JavaScript usage in frequently used apps

In this section we present the insights of our study of *loadUrl* and *evaluateJavascript* on frequently used apps. We collected the most frequently used 196 apps from the Play Store and categorized it into 13 categories. These categories include the most downloaded apps (referred to as Top Apps) and apps that yield high revenues (referred to as Top Gross). Table 6 lists our categories alongside the number of apps studied in each category.

8.1. JavaScript from loadURL *in top 196 apps*

Using *LUDroid*, we prepared a corpus of JavaScript string and objects' fields (containing a JavaScript string, potentially a constant) passed to *loadUrl*. We performed a manual analysis of the corpus and answer the questions: **RQ2.6**, **RQ3.1 - RQ3.3**. We observed that none of the apps in this category injects third-party scripts remotely over an unsecured protocol (**RQ3.1**).

In addition, we also provide insights on the *usage* of *loadUrl* in these apps. Fig. 7 lists the percentage of apps using *loadUrl*. We observed that none of the banking apps we studied use *loadUrl*. In this app corpus we founds 50% usage of *loadUrl* in categories other than Banking. These results suggest that a majority of the popular apps, except those related to security-critical banking tasks, leverage *loadUrl*. In what follows, we describe our insights on the usage of *loadUrl* in each of the categories.

In top-grossing and top apps categories, more than 70% of apps use *loadUrl*. On manual inspection, we observed that the use of *loadUrl* in these apps originates from SDKs. Similar to the benign apps in the last section, either the SDKs pass raw JavaScript or refer to a field variable within the SDK. Approximately 76% of these are mainly related to mobile payments or social networks (**RQ2.6**). Social networking SDKs account for one-third of the usage among these apps. Listing 12 lists the most frequent code we observed in this category of apps. Other categories of SDKs prevalent are advertisement (approx. 10%), e-commerce (approx.

Table 6
Top apps by categories.

Category	#apps studied
Banking	6
Business	10
Education	8
Entertainment	16
Health	10
Online Payments	25
Music	13
News	19
Shopping	17
Social	11
Top Grossing	30
Top Apps	22
Travel	9
Total	196

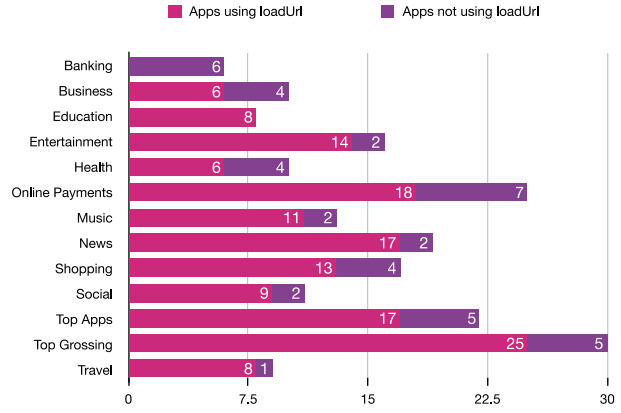

Fig. 7. Apps using *loadUrl* in each category.

5.77%), and also file sharing SDKs such as *DropBox* (approx. 2%). The file-sharing SDK used internally within the *DropBox* app and is used for authentication. These cases highlight the ubiquity of *loadUrl* use among the top downloaded apps and strengthen our results from the benign apps (cf. Section 7.1).

In the education apps, such as *EdX* and *Coursera*, we observed that all apps leverage at least one instance of *loadUrl*. We observed a total of 21 instances of its use in 8 apps. Approximately two-third of the apps leverage *loadUrl* provided by the Facebook SDK **(RQ2.6)** by passing raw JavaScript or accessing the URL through a field defined in the SDK. Fortunately, none of the SDKs were untrusted **(RQ2.6)**. This pattern is similar to those observed in benign apps studied in Section 7.1. We also observed another a similar pattern of resetting the webpage using `about:blank` in one app. A minority (approx. 10%) of these apps pass the parameter to *loadUrl* through inter-procedural function calls. The usage of *loadUrl* in the entertainment category is similar to those in education. Here, also two-third of the apps leverage *loadUrl*. Additionally we observed the usage of the *vungle* ad library, which was also present in the benign apps.

Next, we consider another category: online payments. We observed that 76% of the apps pass object fields to *loadUrl*, while raw JavaScript strings account for the remaining 24%. All of these object fields are from their respective apps. The raw JavaScript is a simple facebook login fallback code which is used in case the Facebook app is not installed on the user's device. This is in contrast with our previous observation that object fields passed to *loadUrl* originate from SDKs.

In social apps, we perceive the use of features that modify an app's look-and-feel through Web APIs. Consider the program fragment in Listing 13, where the app uses Web APIs to modify

Listing 13: Modifying an app's look-and-feel

```
1 (function() {
2   var topbar = document. querySelector('#header[data−sigil=
        "MTopBlueBarHeader"]');
3   topbar.setAttribute('style', 'display:none');
4 }) ()
```

the header section of the app, which is implemented in WebView. Line 2 selects the element named *MTopBlueBarHeader*, and line 3 removes all styling. In this category we found two unique type-2 source code clones, and no instances of passing objects' fields to *loadUrl*.

In other categories, such as shopping and travel, we observed a single type-2 clone of raw JavaScript shown in Listing 12. The same code snippet is most frequent among benign apps as well (Section 7.1). Line 2 initializes a DOM event handler that listens to the event fbPlatformDialogMustClose, and line 3 dispatches the corresponding event.

8.2. Use of evaluateJavascript in frequently used apps

We also analyzed the most frequently used apps from the Google PlayStore to study the JavaScript code passed to the method *evaluateJavascript*. In particular, we found differences between the usage of *evaluateJavascript* and *loadUrl* in these apps. In Section 8.2, we identified some usage patterns of *loadUrl*, many of which similar to the benign apps in Section 7.1. However, in the top 196 apps, our study found a significant difference between the usage of *evaluateJavascript* and *loadUrl*. We found that 26 of these 196 apps use at least one instance of *evaluateJavascript*. These usages contain 158 instances of JavaScript code fragments, forming 16 type-2 code clones. In what follows, we elaborate two cases of frequent JavaScript code in these apps. We also observe an unusual case which – to our surprise – violates basic security practices.

8.2.1. Case study — Controlling page navigation

In our study, we found apps using JavaScript to control page navigation on Android, which is similar to using a regular web page. Consider the code snippet from the payment apps *JioMoney* and *LiquidPay*, where the developers use handlers for keyboard events in JavaScript: `handleBackKey()`, `LoginSubmitClick()`, and document.getElementById('loginIdxSubmitBtn').click(). The first one registers a custom handler for handling the back key on Android devices, while the second one is for logging in via a submit button, and the third one automatically initiates a click event. This case study highlights sophisticated use of *evaluateJavascript* corresponding to the sophisticated use in *loadUrl* (cf. Sections 7.1 and 8.2).

8.2.2. Case study — LiquidPay

Table 7 displays six code snippets we identified using *LuDroid*. The payment app *LiquidPay* uses sensitive information in plain-text in five of its Activities. All of these five Activities use sensitive data such as a password or Bank-ID or perform a sensitive operation such as User Login. The developers include sensitive values in plain-text such as document.getElementById('accesscode').value = "\dots" or document.form1.selBankID.value = "\dots" in *WebView* (cf. Table 7). We have masked the sensitive information by . . . to protect the confidentiality of the information of this app. This case study exemplifies a misuse of *WebView*.

To the best of our knowledge, we did not find any relevant work on static analysis of *evaluateJavascript*. Our study on the

Table 7
JavaScript code passed to *evaluateJavascript*. Sensitive values are masked by . . . to protect the confidentiality of the information in these apps.

Program fragment	%age of apps	Comments
document.loginForm.j_password.value = "..."; document.getElementById('access—pin').value = "..."; document.getElementById('access—code').value = "..."; document.logon1.PASSWORD1.value = "..."; document.form1.selBankID.value = "..."; document.form1.consumerEmail.value = "...";	9 9 9 9 5 2	Injects sensitive data

Listing 14: Malware injecting external results in Android object

```
1 window.HTMLOUT.processHTML( document.getElementById('SearchResults').
      innerHTML);
```

most frequently used apps dataset reveals that developers have found sophisticated use cases for *evaluateJavascript*. In principle *evaluateJavascript* provides an alternative to *loadUrl*. Although *evaluateJavascript* is meant to only evaluate the passed JavaScript expression, it is also possible to mimic the functionality of *loadUrl* by manipulating DOM properties. Again, this makes the existing analyses (Lee et al., 2016; Bae et al., 2019) unsound. A more precise and sound analysis has to also consider the data-flow between Android and *evaluateJavascript*.

9. Evaluation — JavaScript usage in malware

We manually analyzed 1000 malware samples and studied the JavaScript passed to *loadUrl* and *evaluateJavascript*. To our surprise, we did not find any instances of *evaluateJavascript* use in malware. Of all malware samples we studied, 121 samples pass *JavaScript* code to *loadUrl*, while 451 samples use *loadUrl* with an object's field variable. The 121 instances of raw JavaScript constitute eight distinct code clones (type 2) . We further performed a manual analysis of these eight unique JavaScript codes and found two of these JavaScript codes to be similar (type 2 clone) to those in the benign apps. A manual analysis of the field variables again revealed that these originate from SDKs, similar to those we found in benign apps (cf. Table 3).

9.1. Case study — Injecting external objects

We found a usage pattern in 88 of the 121 distinct malware codes (approx 72.73%), where the malware injects results from the WebView into a shared Android object. Listing 14 shows one such instance where the DOM element named *SearchResults*, is passed to the method *processHTML*, and the resulting HTML is accessed through the bridge object *HTMLOUT*. This pattern is similar to injecting external results in Android in Section 7.1. However, in this case, it is more dangerous. By manually searching for the package names, we find approximately 63% of the malware are from repackaged apps. It points out that these apps either have a malicious activity or *WebView*. Having a malicious *WebView* is more dangerous as it potentially exposes it to the Web.

Further, the activities (or WebView) in these apps run with the same privileges as a regular app. For the other 37% apps, we could not find any information on the Google Play Store. Therefore, these apps are taken down by the user, or otherwise, these apps performed some suspicious operation. However, with the current data that we have, it is not possible to give further comments.

Listing 15: Malware susceptible to clickjacking

```
1 function actionClicked(t, u) {
2     var r = prompt('showToast' + JSON.stringify({ method: 'showToast',
          params: (u ? [t, u] : [t]) }));
3     if (r && typeof r === 'string') { return JSON.parse(r).result; }
4 };
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();
```

9.2. Case study — Clickjacking

As another case study we identified is a clickjacking attack. Listing 15 shows a code snippet found in five out of 121 malware (approx. 4.13%). Similar to the case study in Section 7.1, this code snippet is also susceptible to a clickjacking attack. The function *noTapHighlight* in Listing 15 masks user-clicks by removing the feedback given to the users (by removing the highlighted color), thus, making them unaware of the accidental user-interactions with the device. Line 2 of the function *actionClicked* serializes the arguments passed to the method. Line 3 returns the value of the field *result* after a deserialization of the previous statement. We discover the source code in malware, and therefore, the intent of the clickjacking attack could potentially be malicious. However, owing to the limitations of *LuDroid*, we could not ascertain the exact sequence of user-interactions that invokes these functions.

The source code mentioned here is a type-2 clone of the source code in Listing 9, which was also observed in the benign apps in Section 7.1.3. However, it makes the scenario more dangerous than those in benign apps as the same technique is applied to apps with malicious intent. This case study reveals the severity of vulnerable patterns (Section 7.1) if exploited by malicious apps. We attempted to map the respective source codes between benign apps and malware. Unfortunately, a similarity could not be established owing to the limitations of decompilation. Decompilation mangled the variables and class names in many cases, so building a one-to-one mapping between malware and benign apps is not feasible.

9.3. Case study — Analytics SDK in malware

We identify a case study where, to our surprise, we found malware using the analytics SDK *Flurry*. Consider Listing 16 where the function defines an adapter to listen to the Flurry analytics services. The body of the anonymous function implements the functionality possibly required for handling of call requests stored in the call queue. We found this pattern in 8 of 121 apps (approx. 6.61%) of apps.

Listing 16: Malware using SDK

```
(function() {
    var flurryadapter = window.flurryadapter = {};
    flurryadapter.flurryCallQueue = [ ];
    flurryadapter.flurryCallInProgress = false;
    flurryadapter.callComplete = function(cmd) {
        if ( this.flurryCallQueue.length == 0 ) {
            this.flurryCallInProgress = false;
            return;
        }
        var adapterCall = this.flurryCallQueue.pop();
        this.executeNativeCall(adapterCall);
        return "OK";
    };
    //Remaining code from Flurry SDK
}) ();
```

9.4. Case study — SDK usage in malware

We also identified a case where malware uses SDKs, especially those SDKs concerned with advertising. The SDK usage pattern is prevalent in 48.1% of the studied malware samples. Out of these, we found 78.2% (approx. 37% of all malware) originating from SDKs meant for advertising. For example, consider the use of the advertising library *TapJoy* in malware `com.nuttyapps.sally.makeup.salon` and `skloo.mobile.pud`. Note that the former APK is probably derived from a game called "Sally's Makeup Salon", the latter an "Ultimate Drinking" app. Malware is often piggybacked to popular benign software and distributed via non-regulated app stores as a trojan horse in order to be installed voluntarily by naive users. In related work, Lee and Ryu have shown that the TapJoy library performs various sensitive operations. These include launching new activities, retrieving sensor information, and fetching location information and app information (Lee and Ryu, 2019). The last three operations potentially harm user privacy. However, this case is more dangerous because of the malicious intent of the built-in malware. Launching new activities through JavaScript leads to a *App to Web Injection attack* (Hassanshahi et al., 2015; Lee and Ryu, 2019) where the app (in this case, malware) initiates new malicious activities through injected JavaScript.

A closer investigation of which functionality stems from the regular application and in which form the attached malware is beyond the scope of this paper. However, we can see that malware leverages hybrid app activities in order to satisfy its malicious intents and that analyses that aim at analyzing the hybrid communication are bound to provide a precise model of the communication patterns detected in this study. Therefore our work may also serve as a "testbed" for such analyses.

9.5. Comparison of JavaScript usage across datasets

In Section 4.4, we began with three research questions concerning the use of third-party script injection over unsecured protocols **(RQ3.1)**, non-trivial information flows from JavaScript to Android, and third-party libraries using obfuscation in their JavaScript code. Besides, we also discovered the vulnerable use case of leaking private information through clickjacking. In this section, we summarize our observations on the JavaScript strings passed to *loadUrl* for the three categories of apps: Benign, Frequently used, and Malware.

Table 8 presents the behavior observed across the datasets of apps. We observed that (potential) privacy leak is a prevalent behavior of the JavaScript code passed to *loadUrl* in all of the app categories. Non-trivial information flows from JavaScript to Android is the next typical behavior, observed only in the benign apps and malware dataset. Interestingly, the frequently used apps do not show this behavior as they, primarily, use it through field objects (mostly constant strings) of the SDKs. Obfuscated third-parties were observed only in benign apps. Furthermore, frequently used apps and malware were free from third-party script injection attacks. It shows that at least the frequently used apps adhere to basic security practices.

Table 8
Summary of the observed JavaScript behavior in studied apps.

Category	Benign	Frequently used	Malware
Third-party script injection	✓	✗	✗
Non-trivial information Flow	✓	✗	✓
Obfuscation by third-party	✓	✗	✗
Privacy leak	✓	✓	✓

10. Threats to validity

LUDroid was able to derive a plenitude of novel statistics and case studies examining information flows, URL statistics and statistics on JavaScript code. 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 capacitate the design of automatic program analyses that take both sides of the hybrid app into account. Therefore, it cannot be the aim of this work to analyze the plentitude of JavaScript code loaded through HTML files, which requires automatic analysis due to the sheer size of the code base.

Obviously the data we gathered depends on the corpus of apps in the study, and there is a risk that the investigated apps are not representative. However, due to the fact the we randomly chose 7500 apps from a database crawled between 2015 and 2019 should guarantee that we are not biased to any particular app format. To even out the randomness we are adding the most popular apps and malicious apps.

Other threats to the validity of our study stem from the limitation of our approach, which we discuss in the sequel.

10.1. Limitations of LUDroid

LuDroid leverages static analysis techniques and therefore inherits some of the typical challenges and limitations. As a tool that supports manual investigation of bridge communication we are currently not interested to handle challenging topics like native code, reflection, dynamic control flow, obfuscation, and the fact that strings like the URLs passed to *loadURL* can be constructed at runtime. All of these obstacles have been investigated in separate lines of research (Groß et al., 2018; Tiwari et al., 2019; Kan et al., 2019; Li et al., 2016; Grech et al., 2017), and 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 constructed at runtime based on a string parameter. Insights gained in studies that target dynamic code execution (e.g. for JavaScript Richards et al., 2011) are also relevant to understand the semantics of *loadURL* and *evaluateJavascript*. A fully implemented static analysis that automatically derives information flows across the language barrier will eventually want to at least approximate these challenging features.

Native code. In Android applications, it is possible to include native code (e.g., compiled C/C++ code) via the Java Native Interface (JNI). Including assembler semantics into the analysis increases its complexity significantly. Fortunately, we have observed native code rarely during our experiments, so we do not expect a lot of loss in terms of the goals of this study.

Reflection, dynamic control flow and obfuscation. Reflection in Java is a means to access code features at runtime. With reflection, it is possible to make calls and access fields at runtime, dynamically depending on strings and other values derived at runtime. It is, therefore, possible to make the control and data flow entirely dependent on runtime values, therefore eluding static analysis. At present, *LUDroid* does not support analysis for reflective features. Given this limitation, LUDroid will lack insights from apps that intentionally use reflection to prevent static analysis (e.g. trough obfuscation). Again, we have not encountered high usage of reflection in any of the inspected benign and malicious apps.

Limited string analysis. LUDroid requires string analysis to resolve arguments passed to the *loadURL* and *evaluateJavascript* methods. For cases where the string argument is manipulated before being passed, we implemented domain knowledge of the Java *String* class as well as support for the *StringBuilder* class to be able to statically derive the results of straightforward string manipulation. However, LUDroid does not support advanced string manipulations such as array-based string manipulations at the current point of time. More advanced resolution techniques are envisioned for future analyses.

InterProcedural slicing. At the time of this writing LUDroid only supports intra-procedural slicing. For the fast dissemination of typical use cases of bridge communication this design decision was found to be sufficient, as we were not planning to be able to present a complete picture of communication patterns. Future analyses will have to consider more complex communication scenarios to be able to evaluate the security and/or privacy properties of an app under investigation.

11. Related work

Rizzo et al. (2017) 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. (2018) performed a large scale study of the WebView APIs to classify them into four categories of web resource manipulation. Hidhaya et al. (2015) 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. (2017) discovered a new type of WebView-based attack that they call Cross-App WebView Infection (XAWI). Mandal et al. (2018) 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. (2016) 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. (2016) 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, HybriDroid does not provide a comprehensive analysis of hybridization APIs. Their approach is restricted to the fundamental taint analysis of the Information flow from Android to JavaScript and misses other valuable insights. In contrast, we present the full picture of non-trivial data and control flows that occur in Android Web-Hybridization. Besides, we provide experience of the usage of URLs and exemplarily exploit the insecure usage of URLs. Our insights aim to improve the unsoundness in the existing analyses and thus benefit further research.

Neugschwandtner et al. (2013) 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. (2015) 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. (2018) examined so called "Origin Stripping Vulnerabilities" caused by wrongly using the *loadURL* method.

Bae et al. (2019) 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. (2012) 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 and Wagner (2014) 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 (Pokharel et al., 2017; He et al., 2014; Fahl et al., 2012). For example, Pokharel et al. (2017) 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*.

12. Conclusion

In this work, we present a large-scale analysis of *loadURL* and *evaluateJavascript* 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 7500 randomly selected applications from the Google Playstore, the most popular apps and 1000 malware samples. We implemented our semi-automatic analysis approach in a tool called LUDroid that computes the data by using slicing techniques. LUDroid discovered many instances of vulnerabilities, e.g. concerning the usage of unprotected protocols in URLs. To demonstrate the validity of these vulnerabilities we exemplarily showcased the exploitation of one of them. Investigating the most popular apps and malware samples displayed differing characteristics than those found in general benign apps. The insights gained in this study provide valuable input for designing program analyses that are to analyze hybrid Android apps.

CRediT authorship contribution statement

Abhishek Tiwari: Conceptualization, Data curation, Investigation, Methodology, Resources, Software, Validation, Visualization, Writing - original draft, Writing - review & editing. **Jyoti Prakash:** Data curation, Investigation, Methodology, Resources, Software,

Validation, Visualization, Writing - original draft, Writing - review & editing. **Sascha Groß:** Methodology, Software, Validation, Writing - original draft. **Christian Hammer:** Methodology, Investigation, Resources, Validation, Funding acquisition, Supervision, Writing - original draft, Writing - review & editing, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

A. PhoneGap, 2010. PhoneGap native bridge. URL "https://phonegap.com/blog/2010/08/13/phonegap-native-bridge/".

Bae, S., Lee, S., Ryu, S., 2019. Towards understanding and reasoning about Android interoperations. In: Proceedings of the 41st International Conference on Software Engineering. In: ICSE '19, IEEE Press, pp. 223–233. http://dx.doi.org/10.1109/ICSE.2019.00038.

Berners-Lee, T., Fielding, R., Masinter, L., 2004. Uniform Resource Identifier (URI): Generic Syntax. Tech. rep., W3C.

Chin, E., Wagner, D., 2014. Bifocals: Analyzing WebView vulnerabilities in Android applications. In: Revised Selected Papers of the 14th International Workshop on Information Security Applications - Volume 8267. In: WISA 2013, Springer-Verlag New York, Inc., New York, NY, USA, pp. 138–159. http://dx.doi.org/10.1007/978-3-319-05149-9_9.

Chris Klotzbach, D.a.F., Lali Kesiraju, M., at Flurry, A.M., 2018. Flurry state of mobile 2017. URL http://flurrymobile.tumblr.com/post/169545749110/state-of-mobile-2017-mobile-stagnates.

Cortesi, A., Hils, M., 2020. Mitmproxy. https://mitmproxy.org.

Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M., 2012. 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, pp. 50–61.

Fratantonio, Y., Bianchi, A., Robertson, W., Kirda, E., Kruegel, C., Vigna, G., 2016. TriggerScope: Towards detecting logic bombs in Android applications. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 377–396. http://dx.doi.org/10.1109/SP.2016.30.

Genovese, G., 2012. Github. URL "https://github.com/gcesarmza/androidSamples/blob/master/webview/src/main/java/com/gustavogenovese/webview/SynchronousJavascriptInterface.java".

Google, 2018a. Building web apps in WebView. URL https://developer.android.com/guide/webapps/webview.

Google, 2018b. WebView. [cited 12 Sep 2018], URL https://developer.android.com/reference/android/webkit/WebView.

Google, 2018c. WebView for Android. [cited 12 Sep 2018], URL https://developer.chrome.com/multidevice/webview/overview.

Grech, N., Fourtounis, G., Francalanza, A., Smaragdakis, Y., 2017. Heaps don't lie: Countering unsoundness with heap snapshots. Proc. ACM Program. Lang. 1 (OOPSLA), 68:1–68:27. http://dx.doi.org/10.1145/3133892, URL http://doi.acm.org/10.1145/3133892.

Groß, S., Tiwari, A., Hammer, C., 2018. 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. pp. 41–59. http://dx.doi.org/10.1007/978-3-319-98989-1_3.

Gruver, B., 2020. Smali/baksmali. URL https://github.com/JesusFreke/smali.

GS, 2018. Android global market share. http://gs.statcounter.com/os-market-share/mobile/worldwide.

Hassanshahi, B., Jia, Y., Yap, R.H.C., Saxena, P., Liang, Z., 2015. Web-to-application injection attacks on Android: Characterization and detection. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (Eds.), Computer Security – ESORICS 2015. Springer International Publishing, Cham, pp. 577–598.

He, D., Naveed, M., Gunter, C.A., Nahrstedt, K., 2014. Security concerns in Android mHealth apps. In: AMIA Annual Symposium Proceedings, Vol. 2014. American Medical Informatics Association, p. 645.

Hidhaya, S.F., Geetha, A., Kumar, B.N., Sravanth, L.V., Habeeb, A., 2015. Supplementary event-listener injection attack in smart phones. KSII Trans. Internet Inf. Syst. (TIIS) 9 (10), 4191–4203.

Ionic, 2018. Developer survey. URL https://ionicframework.com/survey/2017#trends.

Jin, X., Hu, X., Ying, K., Du, W., Yin, H., Peri, G.N., 2014. 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. In: CCS '14, Association for Computing Machinery, New York, NY, USA, pp. 66–77. http://dx.doi.org/10.1145/2660267.2660275.

Kan, Z., Wang, H., Wu, L., Guo, Y., Xu, G., 2019. Deobfuscating Android native binary code. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. In: ICSE '19, IEEE Press, Piscataway, NJ, USA, pp. 322–323. http://dx.doi.org/10.1109/ICSE-Companion.2019.00135.

Kim, J., Yoon, Y., Yi, K., Shin, J., 2012. Scandal: Static analyzer for detecting privacy leaks in android applications. In: Chen, H., Koved, L., Wallach, D.S. (Eds.), MoST 2012: Mobile Security Technologies 2012. IEEE, Los Alamitos, CA, USA, URL http://ropas.snu.ac.kr/scandal/.

Koschke, R., 2007. Survey of research on software clones. In: Koschke, R., Merlo, E., Walenstein, A. (Eds.), Duplication, Redundancy, and Similarity in Software. In: Dagstuhl Seminar Proceedings, no. 06301, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, pp. 1–21, URL http://drops.dagstuhl.de/opus/volltexte/2007/962.

Lee, S., Dolby, J., Ryu, S., 2016. HybriDroid: static analysis framework for Android hybrid applications. In: Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on. IEEE, pp. 250–261.

Lee, S., Ryu, S., 2019. Adlib: Analyzer for mobile ad platform libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2019, Association for Computing Machinery, New York, NY, USA, pp. 262–272. http://dx.doi.org/10.1145/3293882.3330562.

Li, L., Bissyand'e, T.F., Octeau, D., Klein, J., 2016. DroidRA: Taming reflection to support whole-program analysis of Android apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. In: ISSTA 2016, ACM, New York, NY, USA, pp. 318–329. http://dx.doi.org/10.1145/2931037.2931044, URL http://doi.acm.org/10.1145/2931037.2931044.

Li, T., Wang, X., Zha, M., Chen, K., Wang, X., Xing, L., Bai, X., Zhang, N., Han, X., 2017. 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, pp. 829–844.

Mandal, A.K., Cortesi, A., Ferrara, P., Panarotto, F., Spoto, F., 2018. Vulnerability analysis of Android auto infotainment apps. In: Proceedings of the 15th ACM International Conference on Computing Frontiers. ACM, pp. 183–190.

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

Neugschwandtner, M., Lindorfer, M., Platzer, C., 2013. A view to a kill: WebView exploitation. In: Presented As Part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats. USENIX, Washington, D.C., p. 5, URL https://www.usenix.org/conference/leet13/workshop-program/presentation/Neugschwandtner.

Oltrogge, M., Derr, E., Stransky, C., Acar, Y., Fahl, S., Rossow, C., Pellegrino, G., Bugiel, S., Backes, M., 2018. 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. pp. 102–115. http://dx.doi.org/10.1109/SP.2018.00005, URL doi.ieeecomputersociety.org/10.1109/SP.2018.00005.

Pokharel, S., Choo, K.-K.R., Liu, J., 2017. Can Android VoIP voice conversations be decoded? I can eavesdrop on your Android VoIP communication. Concurr. Comput.: Pract. Exper. 29 (7), e3845.

Rasthofer, S., Arzt, S., Bodden, E., 2014. A machine-learning approach for classifying and categorizing Android sources and sinks. In: NDSS. pp. 259–269. http://dx.doi.org/10.14722/ndss.2014.23039.

Richards, G., Hammer, C., Burg, B., Vitek, J., 2011. The eval that men do. In: Mezini, M. (Ed.), ECOOP 2011 – Object-Oriented Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–78.

Rizzo, C., Cavallaro, L., Kinder, J., 2017. Babelview: Evaluating the impact of code injection attacks in mobile WebViews. arXiv preprint arXiv:1709.05690.

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

Ryszard Wiśniewski, C.T., 2020. Apktool. https://ibotpeaches.github.io/Apktool/.

S. Insight, 2018. Statistics on consumer mobile usage. URL https://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/.

Shehab, M., AlJarrah, A., 2014. Reducing attack surface on Cordova-based hybrid mobile apps. In: Proceedings of the 2nd International Workshop on Mobile Development Lifecycle. In: MobileDeLi '14, Association for Computing Machinery, New York, NY, USA, pp. 1–8. http://dx.doi.org/10.1145/2688412.2688417.

Sun, K., Ryu, S., 2017. Analysis of JavaScript programs: Challenges and research trends. ACM Comput. Surv. 50 (4), 59.

Tiwari, A., Groß, S., Hammer, C., 2019. IIFA: Modular inter-app intent information flow analysis of android applications. In: Chen, S., Choo, K.-K.R., Fu, X., Lou, W., Mohaisen, A. (Eds.), Security and Privacy in Communication Networks. Springer International Publishing, Cham, pp. 335–349.

Tiwari, A., Prakash, J., Groß, S., Hammer, C., 2019. LUDroid: A large scale analysis of Android – web hybridization. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 256–267. http://dx.doi.org/10.1109/SCAM.2019.00036.

Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W., 2017. Deep ground truth analysis of current Android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17). Springer, Bonn, Germany, pp. 252–276.

Weiser, M., 1984. Program slicing. IEEE 10 (4), 352–357.

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

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

Zhang, X., Zhang, Y., Mo, Q., Xia, H., Yang, Z., Yang, M., Wang, X., Lu, L., Duan, H., 2018. An empirical study of web resource manipulation in real-world mobile applications. In: 27th USENIX Security Symposium USENIX Security 18). USENIX Association, pp. 1183–1198.