Part III:
Machine Learning Based Solutions for Security of Mobile Platforms

Chapter 7: Understanding and Mitigating Threats from Hybrid Apps Using Machine Learning (ML)

Chapter 8: Android Web Security Solution Framework Using Cross-device Federated Learning (FL)

Chapter 7: Understanding and Mitigating Threats from Hybrid Apps Using Machine Learning

7.1 Background

The previous part (Part-II) focused primarily on browser-based security using ML. Part III (Chapters 7 and 8) proposes solutions to web security problems related to mobile devices. In this chapter we use conventional Centralized ML for hybrid apps security on mobile platforms. In the thesis, we have used the Android mobile platform for experiments. However, it may be noted that the proposed solution can be applied to other mobile operating systems as well with minor adaptations.

The Android platform has emerged as the most popular computing platform with more than 2.5 billion devicesworking globally, and this platform is rapidly expanding and evolving [173]. These devices include mobiles and tablets, even Android Auto modules in cars, various Android versions running on Televisions, watches, and a host of other smart devices. Application developers have to keep abreast with this ever-evolving ecosystem of Android devices, and competing platforms like iOS and Windows Phone. Android developers are responding to this challenge of supporting multiple platforms through hybrid apps, which allow HTML content to be displayed within an Android app. On the Android platform, hybrid apps use the WebView component to provide the same functionality as web browsers, albeit with complete customizability concerning how and what content is displayed. This allows developers to write platform-neutral code in HTML and JavaScript, which is displayed on any device irrespective of its operating system or its version. Further, it makes updates simpler as the developers merely update the content on the central web server. These capabilities have made hybrid apps an obvious choice for multi-platform applications like Facebook, Twitter, Instagram, etc. Latest survey indicate that hybrid apps constitute approximately

30% of the Android app ecosystem, while the remainder 70% comprises native apps [174]. Moreover, this share of hybrid apps is likely to increase in the coming years.

While convenient, this customized browser component called 'Android WebView', which is used for making hybrid apps, can pose security challenges as the Web Content, in particular JavaScript code, is allowed to interact with Android Application Program Interface (API). Android APIs are software modules used for the development of apps. The code accesses various platform functionalities using this API. Android WebView has a feature in which JavaScript can invoke Java code on the Android platform through a registered interface. While this is extremely useful functionality for hybrid apps, this opens an avenue for malicious exploitation. This offers loopholes for injection attacks that may be mounted by websites being visited, or Cross Side Scripting (XSS) based exploitation, or through man-in-the-middle attack on unsecured HTTP connections (In XSS, malicious JavaScripts are injected into a trusted website by a hacker by exploiting a vulnerability of the dynamic web platform, thereby compromising all browsers that access this website). In this chapter, the security architecture of Android WebView using ML has been analyzed. The assessment of various attributes using software tools and customized Python software has been carried out. This analysis has been carried out on hybrid apps currently hosted on the Google Play store. The study could help identify vulnerable hybrid apps hosted on the store. Such a security analysis of hybrid apps using ML is unprecedented. Highlights of the work presented in the chapter are listed below:

- An Android app named 'WebView Tool' was developed and published on the Google Play store to understand better the WebView component and its exploitation by JavaScript-based injection attacks.
- Identified vulnerable hybrid apps hosted on the Google Play store.
- Corrective measures have been suggested to improve hybrid app security implementations.

- Another Android app named 'WebView Monitor' has been designed and developed to monitor the working of hybrid apps on the Android platform. This app raises alerts during untrusted privilege escalation or when malicious JavaScript is download.

7.2 Related Work and Research Gap

While significant research has been carried out on Android security, very less work has been published on security problems associated with hybrid apps. Most of the work remains confined to the private domain of Google Labs and various cybersecurity and anti-virus companies. Some relevant published work related to the security of hybrid apps is discussed below.

Wagner and Chin were amongst the first to discuss WebView vulnerabilities [175]. They discuss only two types of vulnerabilities - excess authorization and cross side scripting. Other vulnerabilities are not covered in their work. Further, the analysis and modeling of attacks were not covered in depth. Bao et al. have also discussed cross side scripting based attacks on Android hybrid apps [176]. However, the work remained restricted to a particular attack vector and did not cover other security issues of hybrid apps or their mitigating measures. Wei et al. have tried to understand JavaScript vulnerabilities in Android apps [177]. They also proposed a static analysis tool called JSDroid. However, the work remained restricted to static analysis of JavaScript code. Rizzo et al. have discussed a technique of evaluating the impact of code injection attacks in WebViews using flow-based analysis [178]. However, flow-based analysis has lower accuracy compared to ML analysis used in this work. Jianjun et al. [179] and Song et al. [180] have discussed how WebView induces bugs into Android apps. However, they did not include many commonly known JavaScript bugs in their research. Sexton et al. have proposed a model for an end-to-end information flow control for hybrid apps [181]. However, their work was restricted to enforcing confidentiality policy in hybrid apps. Tongxin et al. in their research on cross-app remote infections on mobile WebView (XAWI), have described a unique kind of attack that can manifest when two WebView based apps run on an Android platform and interact with each other [182]. Mohsen et al. have studied JavaScript injection attacks on

WebView-OAuth SDK [183]. WebView-OAuth is a method of authorization that provides login ability to hybrid apps based on a username and password. Their work was limited to this particular attack technique and did not discuss other attack vectors. Imamura et al. have provided a mechanism of WebView access monitoring on Android platform [184]. However, they achieved it by loading WebView with modified source code on the Android platform. Their proposed architecture fails to meet the user aspirations of a simple mechanism that can monitor and alert malicious WebView activities. A simple WebView monitoring app which overcomes the shortcomings of the monitoring app developed by Imamura et al. has been proposed and developed in this chapter.

It is clear from the literature review given above that there is a need to do a comprehensive analysis of hybrid apps security. We have proposed ML solutions for the same in coming sections.

7.3 Understanding Hybrid Apps & WebView Component

7.3.1 Hybrid Apps and Android WebView

Hybrid apps are mobile applications that allow content from websites to be handled and shown on different mobile OS platforms. On the Android platform, hybrid apps use the WebView component to provide web content handling functions akin to a browser. It renders HTML content and runs JavaScript downloaded from the Web Server. Though it is not a full-fledged Web Browser, it is used extensively in Android apps to handle web content as it provides more flexibility than regular browser engines. Popular apps like Facebook, Twitter, and Instagram use WebView for displaying their content. Such apps are known as hybrid apps because they make the best use of both web technology and Android native application framework. These features make hybrid apps extremely popular. It is for this reason that a large number of hybrid apps currently exist on Google Play Store.

It is pertinent to note that WebView is preinstalled in all Android platforms by default. Before Android 4.4 (KitKat), Android WebView used the Webkit engine [185]. Since Android 4.4, the WebView component is based on

the Chromium Open Source Project [186]. The latest WebView component shares the same rendering engine as Chrome, thus making rendering consistent between Chrome Browser and WebView [187]. Till Android version 5.0 (Lollipop), WebView was updated as part of the system update; versions after that have been moved to an APK, allowing it to be updated separately.

7.3.2 Understanding WebView Class

The WebView component is available as a class in Android SDK [188]. The *WebView* Class extends the *AbsoluteLayout* Class and shows web content in the Android app's Activity Layout (Activity Layout provides User Interface to Android app). Android SDK provides two more classes, viz., *WebViewClient*, and *WebSettings*, which are used to customize WebView [189]. While *WebSettings* handles various permissions and settings of WebView, *WebViewClient* is the event handler of WebView that specifies the page navigation behavior of WebView. **Table 7.1** below summarizes the *WebView* Class.

Table 7.1: Summary of the WebView Class

Class	Methods	Description
WebView	*loadData()* *loadUrl()* *loadDataWithBaseURL()*	Renders the Web Data
	getSettings()	Gets the Settings Manager
	setWebViewClient()	Sets the Event Handler
	addJavascriptInterface()	Adds a JavaScript Interface for the interaction of JavaScript Code with Java Code
WebViewClient	*shouldOverrideUrlLoading()*	Web Page Navigation
WebSettings	*setJavaScriptEnabled()*	Enabling JavaScript Execution
	setAllowFileAccess() *setAllowFileAccessFromFileURLs()* *setAllowUniversalAccessFromFile URLs()*	File Access based on various Zones

7.3.3 Development of Hybrid Apps Using WebView Class

WebView is added to an app by either adding the <*WebView element*> in the Activity Layout or by setting the entire Activity Window as WebView in the *onCreate()* function. For loading a URL in WebView, the *loadUrl()* function is called. A sample code snippet below describes this step.

```
WebView myWebView = new WebView(activityContext);
setContentView(myWebView);
myWebView.loadUrl("https://www.bits-pilani.ac.in");
```

The *setJavaScriptEnabled* method of *WebSetting* is used to enable JavaScript on WebView using the code snippet below.

```
WebSettings webSettings = myWebView.getSettings();
webSettings.setJavaScriptEnabled(true);
```

The *setAllowFileAccess(boolean)* method is used to enable or disable file access within WebView. Similarly, *setAllowFileAccessFromFileURLs (boolean)* decides if the JavaScript code running in the context of file scheme URL can access other file scheme URLs, and *setAllowUniversalAccess FromFileURLs(boolean)* determines access of JavaScript to different scheme URLs (file, HTTP, HTTPS).

The *addJavaScriptInterface(Object, String)* method allows JavaScript to interact and run Java Code *Object* attached. This allows JavaScript to invoke functions of the Java *Object*, and through this *Object*, it can call other Java API on the Android platform.

7.3.4 Vulnerabilities: Enabling and Using JavaScript Interface

JavaScript is disabled by default in Android WebView. But, for working with interactive web pages using client-side scripts, JavaScript has to be enabled. This, however, leaves the platform vulnerable to JavaScript-based injection attacks. Frequently, developers, to provide rich functionality in their Android app, use the *addJavaScriptInterface()* method to permit JavaScript interaction with Java code. This functionality enrichment comes at the cost of making the Android platform vulnerable to malicious JavaScript. A malicious JavaScript can use an Android app's permission to cause serious compromise of the complete Android platform. Such vulnerabilities in hybrid apps have been analyzed using ML in the next section.

7.3.5 'WebView Tool' Android App

As mentioned earlier, an Android app named 'WebView Tool' has also been developed and published on Google Play Store [190] to understand better the various facets of WebView and how JavaScript-based injection attacks exploit it. The source code for this app is available on Git Hub [191].

7.4 ML Based Solutions for Hybrid Apps Security

This section describes the use of ML in the security analysis of hybrid apps. In particular, the following tasks have been carried out:

- Identify ML attributes that are good predictors of a vulnerable hybrid apps; thereby identifying characteristics of a vulnerable hybrid app.
- Assess the vulnerability of hybrid apps presently hosted on Google Play.
- Make recommendations for improving the security architecture of hybrid apps based on the characteristics identified through ML.

7.4.1 Methodology

We have used SVM (Support Vector Machine) to analyze and classify hybrid apps on Google Play. Various tools were used for this experiment, and custom code was written in Python to integrate them. SCIKIT-LEARN [168] platform was used for running the SVM Classification. JADX Disassembler [66] was used for disassembling the Android apps. Downloading of relevant app APKs (APK - Android Package Kit, it is the format in which Android apps are distributed) from the Internet and processing of datasets was done using customized Python code.

For classification, a training dataset was required, which comprised of a known list of benign and malicious Android apps. This dataset was prepared from the following sources - Android Malware dataset 2017 (CICAndMal2017) [62], Android Application dataset for Malware Application [192] and Android Anti Malware dataset [193]. As these data sources did not have a common set of attributes, these datasets were processed using customized Python code, and suitable attributes were added and deleted. Attributes not available in these

datasets were extracted after downloading the APKs of these apps from a mirror of Google Play named 'APK Combo' [65]. Google Play does not permit downloading of APKs by bots. Thus, we used a Mirror site that permits downloading using bots. The list of attributes is given in **Table 3.4** of chapter 3. The training dataset prepared from this process comprises 78,767 samples. As the number of malicious apps in these samples are a mere 2.3% of the dataset size, the dataset is skewed. To overcome inaccuracies related to ML in a skewed dataset, oversampling of minority class was carried out using ADASYN [144]. ADASYN-'Adaptive Synthetic' is an algorithm for generating synthetic samples of minority class based on existing minority observations.

For preparing the test dataset, the current hybrid apps available on Google Play were downloaded using the 'APK Combo' mirror site [65]. A total of 34,708 hybrid apps available on Google Play were downloaded for this purpose. These downloaded app APKs were then disassembled using JADX Disassembler [66]. The disassembled code was then parsed using customized Python code to extract relevant attributes for creating the test dataset. It is pertinent to note that the test dataset is unlabeled as we intend to use the trained model to gauge vulnerabilities in the test dataset of live Google Play apps. Further details on the training and test dataset are given in Chapter 3 (preliminary analysis and visualization) and Appendix B (pre-processing code).

After the datasets were prepared, a ML model was trained using the SVM classifier with RBF Kernel. The choice of the SVM classifier was based on its better performance over this dataset vis-a-vis other classifiers like Naive Bayes, Decision Tree, etc. RBF kernel was used with SVM to better model non-linearity in the multi-dimensional data space. As this classification problem has two classes (malicious and benign), the SVM model builds a representation of data points in space separating the two classes with a hyperplane, with the margin made as wide as possible. For data points $[(\overrightarrow{x_1}, y_1),\ldots (\overrightarrow{x_n}, y_n)]$ in the dataset, where, $\overrightarrow{x_i}$ denotes a multi-dimensional point (Twelve-dimensional vector in this dataset), y_i denotes the Class (malicious: -1, benign: 1), and n the size of the training dataset (78,767 samples in this dataset), the hyperplane de-limiting the SVM model is given by equation (7.1),

$$\vec{w}.\vec{x} - b = 0 \qquad (7.1)$$

where, $\vec{w}$ is the normal vector (unit direction vector), and b is a constant depicting the offset of the hyperplane from origin. For a soft margin non-linear data distribution, the SVM classifier tries to maximize the margin by reducing equation (7.2) as given below,

$$\left[\frac{1}{n}\sum_{i=1}^{n} max\left(0,1-y_i(\vec{w}.\vec{x_i}-b)\right)\right]+\lambda\|\vec{w}\|^2 \qquad (7.2)$$

where, λ is a parameter that gives the tradeoff between increasing margin or ensuring $\vec{x_i}$ lies to the correct side of the margin. For handling non-linearity in multi-dimensional space, we used the Radial Basis Function (RBF) kernel as given in equation (7.3),

$$K(x_l, x_m) = e^{-\gamma\|\vec{x_i}-\vec{x_m}\|^2} \qquad (7.3)$$

where γ is a hyperparameter that is tuned with grid search to ensure that our SVM model is well fitted to training data. Python-based SCIKIT-LEARN [168] platform was used for SVM classification. The trained classifier was then run on test dataset to check the vulnerabilities of hybrid apps found on Google Play. The test results are discussed in subsequent sections. Schematic representation of the complete ML process described above is given in **Figure 7.1**.

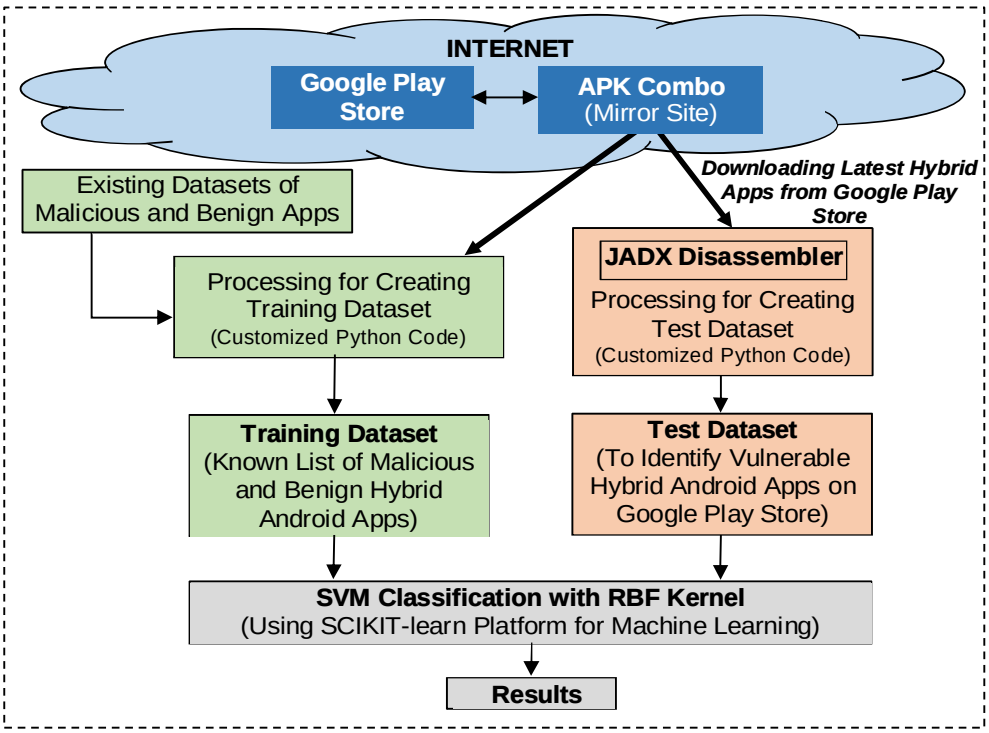

Figure 7.1: Schematic Diagram Describing the ML Process

7.5 Results and Analysis

7.5.1 Training and Validation

The dataset (78,767 samples) was split randomly into training and validation datasets (80% for training- 63014, 20% for validation-15753). The trained SVM classifier was tested for its accuracy by running validation. The hyperparameters were tuned using grid search to avoid overfitting. **Table 7.3** below gives the validation results.

Table 7.2: Validation Results of ML on Hybrid Apps

Metric	Value
Accuracy	0.9986
Recall (TPR, Sensitivity)	0.9976
Specificity (TNR, Selectivity)	0.9996
Precision (PPV)	0.9832
NPV	0.9999
F1 Score	0.9904
*Note: Positive class represents the 'Malicious label'.	

High accuracy of 99.86% vindicates the effectiveness of this model in detecting malicious hybrid apps. Precision, Recall and F1-score are also impressive. The meaning of metrics used in **Table 7.2** has already been given in **Table 4.4**. The confusion matrix is given in **Figure 7.2**. The matrix clearly highlights negligible FNR and FPR during the evaluation of the test dataset.

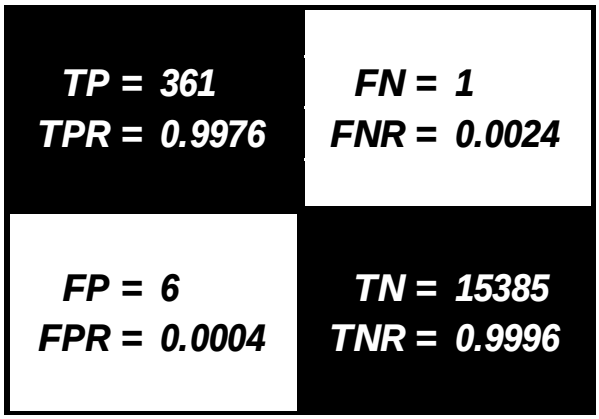

Figure 7.2: Confusion Matrix after Validation

7.5.2 Analyzing Active Samples from Google Play Store

After training and validating the SVM classifier model, the trained model was run on a dataset comprising 34,708 samples from the Google Play store. Results of the classification are given below in **Table 7.4**. As seen, 0.7% of the 34,708 apps downloaded from Google Play were found vulnerable.

Table 7.3: Classification Results- Hybrid Apps on Google Play

Total App Samples in Test Dataset	34,708
Hybrid Apps Found Vulnerable	243 (0.7%)
Hybrid Apps Found Safe	34,535 (99.3%)

7.5.3 Ranking of Attributes

Ranking of attributes was done using the 'Gain Ration Attribute Evaluation' method to find the attributes that best predict vulnerable hybrid apps. In this method, each attribute A_i is assigned a score based on information gain between itself and the class [72] [194]. If A is the attribute and C is the class, equations (7.4) and (7.5) below give entropy H before and after observing the attribute.

$$H(C) = -\sum_{c \in C} p(c)\, log_2\, p(c) \qquad (7.4)$$

$$H(C/A) = -\sum_{a \in A} p(a) \sum_{c \in C} p(c/a)\, log_2\, p(c/a) \qquad (7.5)$$

The above method of ranking attributes was run on the SCIKIT-LEARN platform [168]. Results of this attribute selection are shown graphically in **Figure 7.3**. As can be seen from the ranking in **Figure 7.3**, A3, A4, and A10 are good predictors of hybrid android app's vulnerability. When tested, the top three ranked attributes together gave an accuracy of 98.27%.

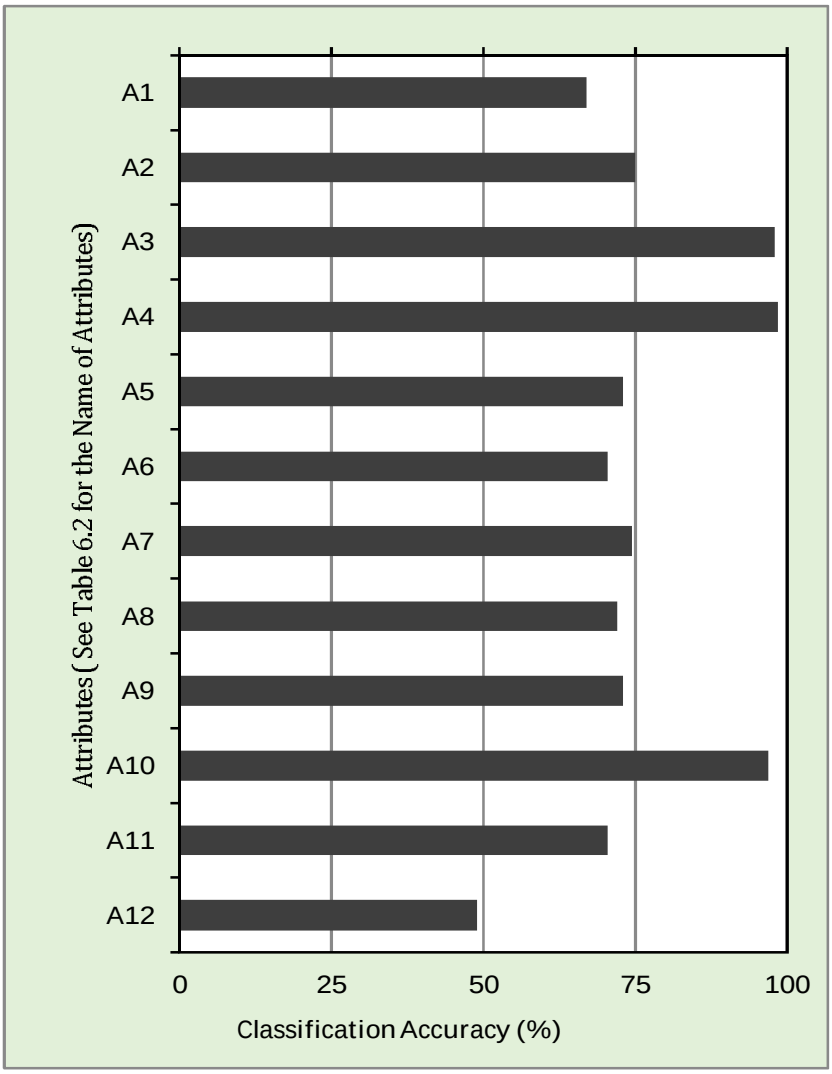

Figure 7.3: Attribute Ranking Based on Classification Accuracy

7.5.4 Interpretation of Results and Inferences

Based on the above analysis, the following aspects can be interpreted and inferred:

A large number of Android hybrid apps are hosted on Google Play store. Few of them have design vulnerabilities, which could not be detected by Google Play Protect. Google Play Protect is a Google Security Program for Android apps. It checks all apps hosted on Google Play Store. Also, it checks Android devices for any potentially harmful app downloaded from other sources. Google Play Protect's inability to detect vulnerabilities is possible because these apps are not malicious ab initio, but are vulnerable to JavaScript-based injection attacks only at runtime.

In this experiment, about 0.7% of Android apps downloaded from Google Play Store were found to be vulnerable. This is a significant number, which cannot be ignored. Thus, there is a critical need to avoid such vulnerabilities by Android developers while writing hybrid apps. Further, Google Play Protect should also incorporate a mechanism to check and identify such potential or active vulnerabilities.

- During experiments, it was also found that Google Play Protect cannot check such vulnerabilities dynamically during the runtime of these hybrid apps. Such a capability is necessary for preventing any runtime exploitation. One such background monitoring app for the Android platform has been developed and discussed in the subsequent section. This app can detect any malicious activity by running dynamic runtime analysis of WebView component of Android platform.

- It emerges from the attribute ranking that attributes A3 (JavaScript_Interface_Defined), A4 (Access_to_ SystemCalls), and A10 (JavaScript_Input_Validation) are the best predictors of vulnerabilities in hybrid app. This means that hybrid apps in which JavaScript Interface is defined and code permits access to system calls are most vulnerable. Further, hybrid apps that have implemented JavaScript input validation methods are benign and safe.

7.6 Recommendations for Improving Security Of Android Hybrid Apps

7.6.1 Recommendations for Android App Developers

Based on the experiments above, the following measures are recommended to be implemented by Android Developers.

- Do not enable JavaScript until it is required.

- If JavaScript is enabled, then write additional code to carry out JavaScript input validation. In the validation process, look for functions that are generally associated with malicious JavaScripts [72], viz., eval(), find(), unescape(), open(). Further, write validation code for

obfuscated JavaScripts as they are more likely to conceal malicious behavior [194]. Validation code should be able to analyze obfuscated JavaScript after carrying out de-obfuscation.

- If JavaScript is enabled, then limit the use of JavaScript Interfaces, which can interact with Android code. Try replacing the dynamic JavaScript Interactions with pure Android code, which can do the same task. This may reduce some flexibility but will significantly improve security.
- When perforce JavaScript access to Android code has to be given, limit access to a specific Class or API. Avoid access to System Calls. Also, do not give access to code or API which has inter app permissions.
- Whereever feasible, use Chrome Custom Tabs [86]. Chrome Custom Tab is a new feature supported on the Android platform since 2015, which permits the launch of Chrome browser from inside the app. Though it lacks the flexibility of WebView, it can be used to create WebView like experience. The added security advantage which it provides is that it uses Google Safe Browsing to protect users from dangerous sites [61]. Google Safe Browsing is a service provided by Google for Chrome Browser, GMail, Google Ads, and Google Search that checks URLs against a known list of malicious URLs before opening the link.

7.6.2 Recommendations for Android OS Platform

Based on the experiments, the following changes or additions are recommended in the Android OS platform.

- There is a need for stricter checking of apps that are being uploaded on Google Play. Especially, apps need to be checked for runtime behavior before being accepted as safe.
- Google Play Protect is presently not capable of monitoring hybrid apps dynamically during their runtime. There is a need for such dynamic monitoring to be implemented on the Android platform. One such app

for dynamic background monitoring has been implemented as part of this work and is discussed in the next section.

- The Android runtime is recommended to be upgraded to permit only limited Android API access to JavaInterfaces. The remaining access should seek explicit user permission before execution.
- It is recommended that Google Safe Browsing API is made mandatory in WebView on the lines of Chrome Tabs [86]. It would then ensure that WebView cannot open sites marked as unsafe by Google Safe Browsing [61].
- Android Runtime is recommended to be upgraded to include a feature wherein Android Code access is limited to the domain, which was initially used in registering the JavaScript Interface. Any redirection away from this domain should be denied by Android Runtime.

7.7 App For Monitoring WebView Vulnerabilities

An Android app named **"WebView Monitor"** has been developed to monitor hybrid apps. This app runs silently in the background and monitors the following WebView related activities:

- Enabling of JavaScript.
- Instantiation of defined JavaScript Interface.
- JavaScript downloaded from the Internet is parsed by this app. During parsing, it looks for signatures of common JavaScript-based injection attacks.

This app is designed to trigger the following alarms:

- An alarm is raised when JavaScript code interacts with Android code.
- An alarm is also raised when JavaScript downloaded by WebView has signatures similar to known injection attacks.

The architecture of the "WebView Monitor" app is given in **Figure 7.4**.

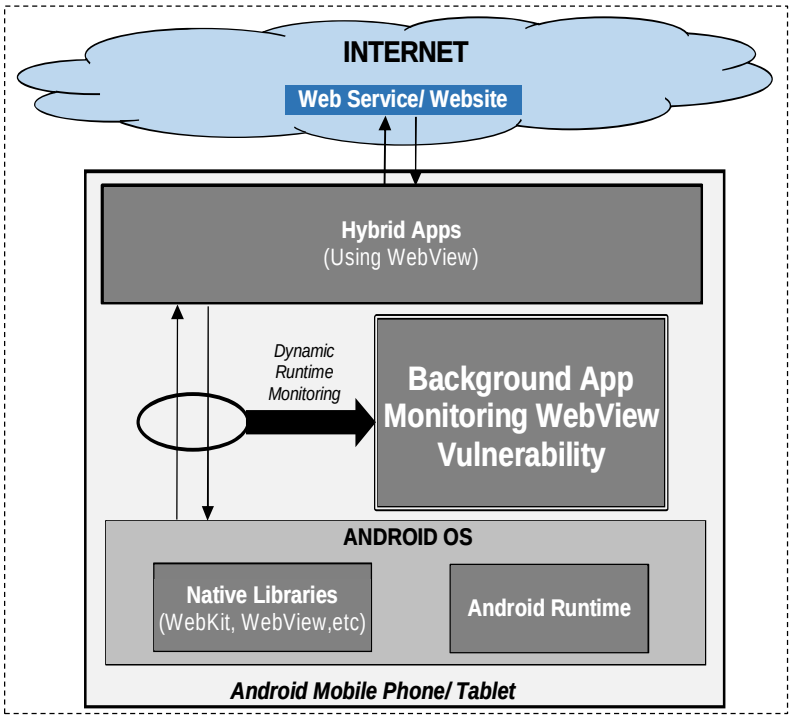

Figure 7.4: The "WebView Monitor" App

The source code of the app is available at Git Hub [195]. The performance of this app was tested by installing and running ten vulnerable Android hybrid apps. During the test, the app performed well and gave alarms for actions that could have exposed the Android mobile to JavaScript-based injection attacks.

7.8 Scope For Future Research

Suggestions which can take the research further are,

- A similar study can be conducted for the iOS platform. The iOS platform is used by mobiles and tablets manufactured by Apple Inc. After Android, it is the second most popular platform globally, based on the number of users. The iOS has an equivalent of the WebView component of Android; it is called 'UIWebView' [196]. The analysis of hybrid apps on the iOS platform using 'UIWebView' can be carried out on similar lines.
- This work had used the 'SVM' for classifying hybrid apps. With recent advances in deep learning, researchers may explore it for the classification of hybrid apps. Further, deep learning has the capability of

carrying out feature extraction from raw disassembled APK code, which may be utilized by the researchers.

- FL [197] has emerged as a popular technology for distributed ML which does not require data from individual sites/mobiles to be uploaded on a central server or cloud. This feature of FL preserves privacy of data as data never leaves the site/mobile. We present a FL model for Android web security in the next chapter.

7.9 Conclusions

The Android platform, by virtue of a large number of users across the globe, is a preferred target of hackers. On the Android platform, hybrid apps are quite popular on Google Play store. Hybrid apps are generally preferred by developers when cross-platform services have to be provided simultaneously on the web and various mobile OS platforms like Android, iOS, etc. While many hybrid apps exist on Google Play store, and many are being developed, not much work has been done to study hybrid apps security holistically. This work is an initiative in this direction. In this chapter, we have described the security architecture of hybrid Android apps. The Android WebView component used in hybrid apps has been discussed in detail, bringing out its security vulnerabilities. An Android app named 'Web View Tool" has been developed to facilitate a better understanding of WebView Security. Apart from this, we have analyzed various JavaScript-based attacks on Android hybrid apps and have used a ML model to detect them with high accuracy. Based on this analysis and experimental findings of the ML process, we have recommended threat mitigation methodologies. As assessed by carrying out testing on random samples, these mitigation techniques can reduce the vulnerabilities significantly. Also, we have developed an Android app that can monitor and detect malicious activities of hybrid apps. The study carried will help us better understand hybrid app security and facilitate identifying vulnerabilities. Further, it will help us monitor hybrid apps and prevent them from indulging in any malicious activity.