Appendix A: Webpages Dataset

The details of experimental setup and code for reproducing the webpages dataset is given in this Appendix.

The dataset was collected by scraping websites across the globe on the Internet. MalCrawler [67], a special purpose focused crawler, was used for this task (refer to Chapter 2 for more details on MalCrawler). MalCrawler [67] is a preferred crawler for this task as it seeks more malicious websites than a random crawl by any other generic web crawler. Further, it is a uniquely designed crawler that does not get entangled in deep crawls or dynamic websites. The data collected from the crawl was then processed to extract the attributes, which have been described in Chapter 3. The basic information captured during the crawl included IP address, URL, and web content. Other attributes were thereafter extracted using customized Python Code. The choice of attributes extracted for this dataset was based on its relevance in malicious webpage classification, as brought out by Singh et al. in their paper [72] (refer to chapter 4 for more details on attribute selection). The attribute 'url_len' was computed from 'url' using the Python code given in **Figure A.1**.

```
import pandas as pd

# Loading the raw data file into
Pandas Dataframe
df = pd.read_csv("raw_data.csv")

#Generating 'url_len' from 'url'
df['url_len']= df['url'].str.len()
```

Figure A.1: Code Snippet for Extracting 'url_len'

The 'geo_loc' attribute, which gives out the country to which the IP Address belongs, is computed from GeoIP Database [73], as given by the code in **Figure A.2**.

```
        # Loading the GeoIP Database
reader = geoip2.database.Reader('GeoLite2-Country.mmdb')

def geoloc(ip_add): #Function for Computing geo_loc
   geo_loc=""
   try:
      response = reader.country(ip_add)
      geo_loc = response.country.name
   except Exception as msg:
      geo_loc= ""
   return geo_loc

#Fill the 'geo_loc' column in df
   for x in df.index:
      df['geo_loc'][x]= geoloc(df['ip_add'][x]
```

Figure A.2: Code Snippet for Computing 'geo_loc'

Attribute 'js_len' is computed using the code given in **Figure A.3**. The JavaScript code, enclosed within '<script>*****</script>' tags are identified and extracted using regex function.

```
import re      #importing regex for string selection and parsing

def get_js_len_inKB(content): #Function for computing 'js_len from Web
Content
   js=re.findall(r'<script>(.*?)</script>',content)
   complete_js=''.join(js)
   js_len = len(content.encode('utf-8'))/1000
   return js_len

for x in df.index: #Computing and Putting 'js_len' in Pandas Dataframe
```

Figure A.3: Code Snippet for Computing 'js_len'

Attribute 'js_obf_len' requires decoding of the obfuscated JavaScript code before computation. This decoding of obfuscated code is carried out using 'JavaScript Auto De-Obfuscator' (JSADO) [74] and Selenium Python library [75]. Code for de-obfuscation is available at [76]. Attribute 'tld' is computed from URL using the Python 'Tld' library [77]. Code snippet for this extraction is given below in **Figure A.4**.

```
from tld import get_tld   # Importing the tld library

for x in df.index:
   df['tld'][x]          =          get_tld(str(df['url'][x]),
```

Figure A.4: Code Snippet for Extracting 'tld'

Attribute 'who_is' is computed with the WHOIS API [78] using the code snippet shown below in **Figure A.5**.

```
from urllib.request import urlopen     # Importing url library
import json                                  # Importing the JSON Module

url = 'https://www.bits-pilani.ac.in' #A sample URL
apiKey = 'at_YC7W9LM2w1lQOCMmN0KUe3OU7B8Jc'
url = 'https://www.whoisxmlapi.com/whoisserver/WhoisService?'\
  + 'domainName=' + url + '&apiKey=' + apiKey + "&outputFormat=JSON"

whois_data= urlopen(url).read().decode('utf8') #WHO IS info returned by
API
data=json.loads(whois_data) # Converting it from JSON to a Python Dict
Object
if data['registrarName']=="":
  who_is = 'incomplete'
else:
  who_is = 'complete'
```

Figure A.5: Code Snippet for Computing 'who_is'

Attribute 'https' is computed using the code shown in **Figure A.6** below.

```
import http.client # Import http client library

for x in df.index:
  https_status= False
  try:
    conn = http.client.HTTPSConnection(df['url'][x])
    conn.request("HEAD", "/")
    res = conn.getresponse()
    if res.status == 200 or res.status==301 or
res.status==302:
      https_status= True
  except Exception as msg:
    print(x,"Error: ",msg)
  finally:
```

Figure A.6: Code Snippet for Computing 'https'

Class labels for this dataset have been generated using the Google Safe Browsing API (refer to the sample code for generating labels below in **Figure A.7**).

```
KEY=
"AIzaSyABO6DPGmHpCs8U5ii1Efkp1dUPJHQfGpo"

s = SafeBrowsing(KEY)

for x in df.index:
  try:
    url = df['url'][x]
    r = s.lookup_urls([url])
    label=r[url]['malicious']
    df['label']=label
```

Figure A.7: Code Snippet for Class Labels

The code used for generating and pre-processing this dataset has been hosted online on the Mendeley repository [68] and Kaggle [79] to facilitate future research.

Appendix B: Hybrid Apps Dataset

The details of experimental setup and code for reproducing the hybrid apps dataset is given in this Appendix.

The dataset was compiled by collecting data from various datasets and downloading and disassembling numerous Android APKs from multiple sources using the JADX disassembler [87]. Datasets that were used include the Android Malware dataset 2017 (CICAndMal2017) [62], Android Application dataset for Malware Application [80], and Android Anti Malware dataset [81]. As these data sources did not have common attributes, these datasets were processed using customized Python code, and suitable attributes were added and deleted. Most attributes were not available in these datasets. Thus, they were extracted from APKs downloaded from a mirror of Google Play named 'APK Combo' [65], as Google Play does not permit downloading APKs by bots. For Disassembling the hybrid apps downloaded from the 'APK Combo' mirror, JADX Disassembler [87] was used.

The code for disassembling Android APKs using JADX disassembler is given below in **Figure B.1**. The entire code (in Java) for using JADX disassembler for disassembling APKs during the pre-processing step is hosted on Github [84].

```
private static int processAndSave(JadxArgs jadxArgs) {
  jadxArgs.setCodeCache(new NoOpCodeCache());
  jadxArgs.setCodeWriterProvider(SimpleCodeWriter::new);
  try (JadxDecompiler jadx = new JadxDecompiler(jadxArgs)) {
   jadx.load();
   if (LogHelper.getLogLevel() ==
  LogHelper.LogLevelEnum.QUIET) {
    jadx.save();
   } else {
    jadx.save(500, (done, total) -> {
     int progress = (int) (done * 100.0 / total);
     System.out.printf("INFO  - progress: %d of %d (%d%%)\r",
  done, total, progress);
    });
   }
   int errorsCount = jadx.getErrorsCount();
  }
  return 0;
  }
```

Figure B.1: Code Snippet for Disassembling APKs Using JADX Disassembler

The Python code for checking usage of either Chrome Custom Tabs [86] or WebView is given in **Figure B.2**. The code looks for instantiation of the Chrome Custom Tab within the Android Code by searching for *'CustomTabsIntent.Builder'*, which is the code for instantiating it [88].

```
#  Checked by looking for substring 'CustomTabsIntent.Builder'
# This substring is used to instantiate a
# Chrome Custom Tab within Android Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("False")
    else:
        print("True")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = "CustomTabsIntent.Builder""
```

Figure B.2: Code Snippet for Checking WebView or Chrome Tab Usage

The Python code for checking whether JavaScripts are enabled for the app is given in **Figure B.3**. This code looks for a *WebView* setting named *'setJavaScriptEnabled(True)'* in the disassembled app code obtained from the output of the JADX disassembler (refer to **Figure B.1**). The presence of this setting confirms enabling of JavaScript within the *WebView* context [89].

```
#  Check for SetJavaScript Enabled settings in Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("False")
    else:
        print("True")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = "webSettings.setJavaScriptEnabled(true)"
check(disaasembed_code, Check_string)
```

Figure B.3: Code Snippet for Checking Whether JavaScript is Enabled or Not

The Python code for checking whether the JavaScript Interface was defined in the app is given in **Figure B.4**. It is confirmed by looking for the annotation *'@JavascriptInterface'* in the disassembled Android code [90]. This annotation has to be present before a public method in the Android code to become accessible to JavaScript on a webpage during runtime.

```
#  Check for '@JavascriptInterface' in Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("False")
    else:
        print("True")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = "@JavascriptInterface"
```

Figure B.4: Code Snippet for Checking JavaScript Interface

The Python code for checking whether access to system calls is permitted or not in the app is given in **Figure B.5**. Access to system calls in an app can be allowed by providing specific permissions in the '<*user-permission*>' attribute of the *'AndroidManifest.xml'* file [91]. These permissions, which can facilitate access to system calls, are listed in the *'matches'* variable in **Figure B.5**. The code below looks for a match of these permissions in the *'AndroidManifest.xml'* file extracted from the disassembled app code.

```
import xml.etree.ElementTree as ET
#Read the AndroidManifest.xml file from disassembled code
root = ET.parse("AndroidManifest.xml").getroot()
permissions = root.findall("uses-permission")
# Permissions  that  can start access to a system call
matches = ["REQUEST_COMPANION_RUN_IN_BACKGROUND ",
"REQUEST_INSTALL_PACKAGES ", "
SIGNAL_PERSISTENT_PROCESSES", "INSTALL_PACKAGES "]

for perm in permissions:
   for att in perm.attrib:
      if any(x in perm.attrib for x in matches):
         print("True")
      else:
```

Figure B.5: Code Snippet to Check Access to System Calls

Python code to check *WebView* permission for obfuscated JavaScript code is given in **Figure B.6**. While *WebView* does not have a mechanism for explicitly denying obfuscated JavaScript code, it can accomplish this through two techniques. First, it can deny JavaScript altogether using the setting' *setJavaScriptEnabled(True)' as* described through the code in **Figure B.3**; however, it is not a preferred method as it disables even clear JavaScripts. Second, it can search for obfuscated code and deny it from running. This search for obfuscated code is generally done by checking for *'eval()'* and *'regexp()'*

functions in the code [92]. Code snippet which identifies apps in which *WebView* uses the second technique is given in **Figure B.6**.

```
#Read the disassembled code and look for the keywords
# Keywords to be matched
matches = ["eval() ", "regexp"]

disassembled_code = outputOfJADX # Feed the output of JADX here

if any(x in disassembled_code for x in matches):
    print("True")
else:
    print("False")
```

Figure B.6: Code Snippet to Check Permission for Obfuscated JavaScript

The Python code to check whether the Android code in *JavaScriptInterface* is obfuscated or not is given in **Figure B.7**. Generally, hackers would like to obfuscate Android code that is there in the *JavaScriptInterface*. **Figure B.7** brings out the code to identify such obfuscated Android Code. It uses the trick that any obfuscated Android code in the *JavaScriptInterface* will have special characters; thus, it looks for such special characters in the code block.

```
#Extract the JavaScript Interface Code from disassembled code
disassembled_code = outputOfJADX # Feed the output of JADX here
# Define the start & end index of Android Code within the JavaScript Interface
start = disassembled_code.find("@JavascriptInterface") +
len("@JavascriptInterface")
end = s.find(";}")
android_interface_code = s[start:end] #This will give the Android Interface code

special_characters = "!@#$%^?~"
 #These special characters are generally found in obfuscated code

if any(c in special_characters for c in android_interface_code):
   print("True")
```

Figure B.7: Code Snippet to Check for Obfuscation of JavaScript Interface Code

Figure B.8 gives the code to check whether *WebView* allows access to URLs apart from the Web Server being visited. To open outside URLs that the user clicks, the *WebView* instance must be provided with a *WebViewClient* instance using the *setWebViewClient()* method [93]. Thus, the presence of the *WebViewClient* instance can indicate this behavior.

```
#Get the disassembled code
disassembled_code = outputOfJADX # Feed the output of JADX here

match = "WebViewClient" # Defining the string to be searched

if any(x in match for x in disassembled_code):
   print("True")
else:
   print("False")
```

Figure B.8: Code Snippet to Check Permission for Outside URLs

The code to check whether the app uses Google Safe Browsing or not is given in **Figure B.9**. The android platform can allow the use of Google Safe Browsing API [61] by *WebView*. This feature is enabled by the *'setSafeBrowsingEnabled()'* setting in the *WebView* code. The presence of this setting is used to identify whether the Google Safe Browsing feature is being used or not.

```
#  Check for 'setSafeBrowsingEnabled()' in Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("False")
    else:
        print("True")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = "setSafeBrowsingEnabled()"
check(disaasembed_code, Check_string)
```

Figure B.9: Code Snippet to Check Whether Safe Browsing is Being Used or Not

The code to check whether the app uses HTTP or HTTPS to connect to the server is given in **Figure B.10**. *WebView* uses the *loadUrl()* method for loading URLs [94]. By checking the URLs being used in *loadUrl()*, the usage of HTTPS can be ascertained.

```
#  Check for 'HTTPS' in loadUrl() method of Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("True")
    else:
        print("False")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = "loadUrl("https://"
check(disaasembed_code, Check_string)
```

Figure B.10: Code Snippet to Check Usage of HTTPS

Figure B.11 gives the code to check whether the app is using JavaScript validation or not. While *WebView* does not have a predefined function for validation, it can validate using the *evaluateJavascript()* method of the *WebView* class [95]. This trick is used to check if JavaScript validation is being carried out by *WebView* or not.

```
# Check for ‘evaluateJavaScript() in Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("True")
    else:
        print("False")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = “evaluateJavaScript”
check(disaasembed_code, Check_string)
```

Figure B.11: Code Snippet to Check the Usage of JavaScript Input Validation

Python code to check whether the app permits web redirection is given in **Figure B.12**. Whenever Android *WebView* has to load a redirected URL, it has to override the *shouldOverrideUrlLoading()* method [96]. The presence of this method in the code is an indication that *WebView* is permitting the usage of web redirections.

```
# Check for ‘shouldOverrideUrlLoading() in Disassembled Code

def check(string, sub_string):
    if (string.find(check_string) == -1):
        print("True")
    else:
        print("False")

disassembled_code = outputOfJADX # Feed the output of JADX here
Check_string = “shouldOverrideUrlLoading”
check(disaasembed_code, Check_string)
```

Figure B.12: Code Snippet to Check Whether Web Redirection is Permitted or Not

Figure B.13 gives the code to compute the length of the Android code that is defined in the JavaScript Interface of WebView.

```
#Extract the JavaScript Interface Code from disassembled code
disassembled_code = outputOfJADX # Feed the output of JADX here
# Define the start & end index of Android Code within the JavaScript
Interface
start = disassembled_code.find("@JavascriptInterface") +
len("@JavascriptInterface")
end = s.find(";}")
android_interface_code = s[start:end] #This will give the Android
Interface code

#Compute length of android_interface_code
def utf8len(s):
   return len(s.encode('utf-8'))/1000 #Length of string in KB
```

Figure B.13: Code Snippet for Checking Android Code Length in JavaScript Interface

It may be noted that the Class Label was created using information about the app being malicious or not, which was collected from multiple sources and datasets [62][80][81] already available on the Internet.

The code used for generating and pre-processing this dataset has been hosted online on GitHub to facilitate further research [84][85].

Appendix C: Details of Software Libraries and Tools Used

The details of software libraries and tools used for this thesis are given below in alphabetical order.

Adversarial Robustness Toolbox (ART) Library. ART [236] is a Python library for ML security. It provides tools to evaluate, defend and verify ML models and applications against adversarial threats. It supports Keras and TensorFlow libraries (described later) and can be used along with them. In this thesis, the ART library has been used for strengthening the FL model against adversarial attacks (refer to Chapter 8).

Android SDK. Android Software Development Kit (SDK) [242] is the set of software development tools and libraries used to develop Android Applications. In Chapter 7 of this thesis, Android SDK has been used to develop two apps, viz., 'WebView Tools' and ' WebView Monitor'. In Chapter 8, it has been used to develop the FL client app for the Android platform.

Android Studio. Android Studio [243] is the Integrated Development Environment (IDE) for the Android operating system. This IDE was used to develop and test all Android apps that were created for this thesis.

GeoIP Database. GeoIP database [73] provides the geographical mapping of IP addresses. Using the GeoIP API, the country and city of an IP address can be determined. In this thesis, it has been used for determining the geographical location of webpages during the dataset preparation and pre-processing stage (refer to Chapters 2 and 3).

Google Colab ML Platform. Google Colaboratory (Colab) [244] is the online platform for writing and executing Python code. It provides cloud service based Jupyter Python notebooks. It supports various ML libraries and is thus is used extensively for running ML Python code. It supports both CPU and GPU. This platform has been used for running the FL simulation code in Chapter 8.

Google Safe Browsing API. Google Safe Browsing API [61] is a Google service that permits the checking of URLs against a constantly updated list of unsafe web sources. It has been used in this thesis for generating Class Labels of webpages dataset for supervised ML (refer to chapters 2 and 3).

HTML Unit Browser Emulation Library. The HTML Unit [53] is a browser emulation library based in Java. It is used to emulate a Browser session. Certain features can be tested only by emulating a Browser session, e.g., redirection, cloaking, etc. It has been used for such testing in Chapters 2 and 4 of this thesis.

JADX APK Disassembling Library. JADX [66] is an Android APK disassembling library. Given the dex code of an Android app, it produces the Java source code of that app. In this thesis, it was used for disassembling Android apps while preparing the hybrid apps dataset (refer to Chapters 2, 3, and 7).

JavaScript Auto De-Obfuscator (JSADO) Library. 'JavaScript Auto De-Obfuscator' (JSADO) [74] is a JavaScript de-obfuscation library. In the thesis, it has been used for de-obfuscating the JavaScript at the feature extraction stage (Refer to Chapters 2, 3, 5 and 6).

JSoup Parsing Library. The JSoup library [52] is a Java-based library with web page parsing capability. This library has been used for parsing web pages and extracting - hyperlinks, document content, and JavaScript tags during the pre-processing stage (Refer to Chapters 2 and 3).

JupyterLab IDE. JupyterLab [245] is a web-based Interactive Development Environment (IDE) for Jupyter Notebooks. Jupyter Notebooks are used for running the Python code and visualization. In this thesis, all Python code used for ML has been written in Jupyter Notebooks in JupyterLab IDE.

Kaggle ML Platform. Kaggle [246] is a cloud-based ML platform hosted by Google for publishing datasets, building models, running and sharing them with the community of data scientists. In this thesis, it has been used for

publishing the datasets and ML code developed as part of the work to facilitate further research in the field of web security.

Keras ML Library. Keras [148] is an open-source deep learning library in Python. It acts as an interface for the TensorFlow library (refer to C.20 in this Appendix). Apart from supporting ANN, it also supports CNN and RNN. It supports utilities like dropout, normalization, etc. It supports CPU, GPU, and TPU for ML training. In this thesis, it has been used for making the DNN models discussed in Chapters 5, 6, and 8.

NetBeans IDE and Profiler. NetBeans [247] is an Integrated Development Environment (IDE) for Java. NetBeans has been used in this thesis for developing all Java-based software within the scope of this thesis. For example, it was used to develop the MalCrawler, which has been written in Java (refer to Chapter 2). NetBeans also has a Profiler module that can measure the CPU cycles and memory utilization while running the Java code. This profiler has been used in Chapter 4 for profiling the attribute extraction and pre-processing step (to compare the attributes).

NumPy Vector Calculus Python library. NumPy [248] is a Python library that provides functions for vector calculus, like handling of multi-dimensional arrays and matrices. This library has been used for writing various ML codes in this thesis.

PostgreSQL Database. PostgreSQL [249] is an open-source relational database management system that is SQL compliant. In this thesis, it has been used for storing data during the data collection phase (refer to MalCrawler in Chapter 2).

Rhino JavaScript Emulation Library. Rhino [54] is a Java-based library that can run JavaScript. In this thesis, it has been used to run JavaScript in a sandboxed environment for analyzing runtime behavior (refer to Chapters 2 and 4).

Seaborn Python Visualization Library. Seaborn [250] is a Python data visualization library based on Matplotlib. It has been used on various occasions in this thesis for plotting the ML results (refer to Chapters 5, 6, and 8).

Scikit-learn ML Library. Scikit-learn [168] is an open-source ML library in Python. In this thesis, it has been used for building Conventional ML models (refer to Chapter 4) and also for grid search of hyperparameters

TensorFlow ML Library. TensorFlow [147] is an open-source ML library in Python that was released by Google. It supports deep learning. In this thesis, it has been used along with Keras (refer to C.13) to build DNN models and LSTM Autoencoders (refer to Chapters 5 and 6).

Tensor Flow Federated (TFF) FL Library. TFF [240] is an extension of the TensorFlow library that supports FL models and tasks. In this thesis, it has been used for building and training the FL model for Android web security (refer to Chapter 8).

Tensor Processing Unit (TPU). Tensor Processing Unit (TPU) [251] are Application Specific Integrated Circuits (ASIC) developed by Google for high-speed deep learning. It supports TensorFlow (refer C.20). These TPUs are accessible for ML training through the Kaggle platform (refer to C.12) or through the Google AI cloud. In this thesis, TPUs have been used for speeding up ML training (refer to Chapters 5 and 6).

TensorBoard ML Visualization Library. The TensorBoard library [252] provides storage, retrieval, visualization of ML results produced using TensorFlow (refer C.20). It has been in this thesis for plotting of ML results.

WEKA Data Mining Library. Waikato Environment for Knowledge Analysis (WEKA) [55] is a Java-based Data Mining library. In this thesis, it has been used for URL prediction tasks as part of MalCrawler (refer to Chapter 2) and also for Conventional ML based classification (refer to Chapter 4).

WHOISXML API for DNS Information. The WHOISXML API [78] provides the Whois DNS information for domains and IP addresses. In this thesis, it has been used for extracting domain-related information (like

ownership records and registration details) of webpages collected from the Internet (refer to data collection and pre-processing in Chapters 2 and 3).