Connection Science

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/ccos20

Effective classification of android malware families through dynamic features and neural networks

Gianni D'Angelo, Francesco Palmieri, Antonio Robustelli & Arcangelo Castiglione

To cite this article: Gianni D'Angelo, Francesco Palmieri, Antonio Robustelli & Arcangelo Castiglione (2021) Effective classification of android malware families through dynamic features and neural networks, Connection Science, 33:3, 786-801, DOI: 10.1080/09540091.2021.1889977

To link to this article: https://doi.org/10.1080/09540091.2021.1889977

Published online: 23 Feb 2021.

CONNECTION SCIENCE
2021, VOL. 33, NO. 3, 786–801
https://doi.org/10.1080/09540091.2021.1889977

Effective classification of android malware families through dynamic features and neural networks

Gianni D'Angelo, Francesco Palmieri, Antonio Robustelli and Arcangelo Castiglione

Department of Computer Science, University of Salerno, Fisciano, SA, Italy

ABSTRACT

Due to their open nature and popularity, Android-based devices have attracted several end-users around the World and are one of the main targets for attackers. Because of the reasons given above, it is necessary to build tools that can reliably detect zero-day malware on these devices. At the moment, many of the frameworks that have been proposed to detect malware applications leverage Machine Learning (ML) techniques. However, an essential requirement to build these frameworks consists of using very large and sophisticated datasets for model construction and training purposes. Their success, indeed, strongly depends on the choice of the right features used for building a classification model providing adequate generalisation capability. Furthermore, the creation of a training dataset that well represents the malware properties and behaviour is one of the most critical challenges in malware analysis. Therefore, the main aim of this paper is proposing a new dataset called *Unisa Malware Dataset (UMD)* available on http://antlab.di.unisa.it/malware/, which is based on the extraction of static and dynamic features characterising the malware activities. Additionally, we will show some experiments concerning common ML tools to demonstrate how it is possible to build efficient ML-based malware classification frameworks using the proposed dataset.

ARTICLE HISTORY
Received 14 November 2020
Accepted 5 February 2021

KEYWORDS
Malware analysis; malware dataset; machine learning; convolutional neural network; recurrent neural network.

1. Introduction

Android-based devices have recently attracted numerous end-users around the World, and today the Android operating system is one of the main targets for malware threats. There are currently many types of unprotected Android devices, such as smartphones, tablets, smart TVs, wearable devices (e.g. Google glasses), and so on. At the same time, a huge number of apps become available on third-party marketplaces that often do not implement any pre-admission malware check. A study of only a few years ago (Symantec, 2018), reported an average of about 38,000 new malware samples detected per day over. Again, the number of mobile app downloads observed worldwide is expected to reach 352.9 billion in 2021 (Statista, 2019). According to Ericsson mobility report (Cerwall, 2015), by 2020, smartphones' subscription will be more than 6 billion, and mobile devices will generate 80% percent of network traffic.

CONTACT Gianni D'Angelo giadangelo@unisa.it

With the rapid growth of malware technologies targeted explicitly on the Android platform (Potha et al., 2020), researchers are attempting to hinder them by proposing more effective malware analysers able to operate at runtime or before launching an application. Different static and dynamic approaches have been proposed to extract representative behavioural information of Android apps and develop models capable of detecting malicious apps and identifying their type. These methods are often combined with Artificial Intelligence (AI), like Machine Learning (ML) or Deep Learning (DL) techniques (Tian et al., 2020), because they can learn complex patterns, features, and relationships from huge amounts of data coming from the observation of past experiences (Ogiela, 2010). In particular, we remark that in the last decade, DL techniques have played an increasingly important role in malware classification (Ficco, 2019, june; Karbab et al., 2018; Martín García et al., 2018). However, such techniques require a large number of samples. One of the most critical malware analysis challenges is creating a dataset that represents the malware's characteristics and behaviour to be used for model training and knowledge construction. For example, Lashkari et al. (2018) proposed a systematic approach to generate Android malware datasets by using real smartphones instead of emulators and develop a new dataset called *CICAndMal2017*.

This paper's main aim is to propose a new dataset, called *Unisa Malware Dataset (UMD)*, based on the extraction of static and dynamic features characterising the malware program activity. We will also show some experiments based on the use of common ML end DL techniques to demonstrate how it is possible to build efficient malware classification solutions by using the proposed dataset for properly training several kinds of AI-based models.

The rest of the paper is organised as follows: Section 2 will present an overview of related works about malware classification methods for Android devices. Section 3 will describe the static and dynamic features extracted to build the UMD dataset. Section 4 will present the CuckooDroid sandbox used to extract the static and dynamic features. Section 5 will present an overview of the proposed approach adopted to perform our experiments. Section 6 will show the steps about the dataset definition. Section 7 will present the results of our experiment based on the proposed dataset. Finally, Section 8 will show the conclusions and future works.

2. Related works

Nowadays, many detection frameworks and methodologies based on static and dynamic analysis have been proposed (Chan & Song, 2014; Reina et al., 2013; Taheri et al., 2019) to counter the continuous growth and diffusion of Android malware. However, it is possible to find other important detection approaches. Static approaches analyse the program structure to acquire a clear view of its behaviour by performing reverse engineering, hence inferring a database of signatures useful to identify known attacks. For example, Zhang et al. (2014) proposed a novel semantic-based approach for classifying Android malware by using dependency graphs that were created from a static analysis of the application code. Again, Onwuzurike et al. (2019) presented MaMaDROID, a new malware detection solution for Android that is based on static analysis and checks the sequences of API calls associated with the activity of an application, to map them to their packages and families. However, static approaches are adversely affected by the use of obfuscation techniques, and also, they are ineffective against polymorphic malware. In fact, this malware family

always adopts a transformation engine generating new decryption routines for the malicious code to be executed without the need to modify the code itself. In this way, the main feature of polymorphic malware is making any signature-based detection technique practically ineffective.

This is the reason why, recently, the approaches based on dynamic analysis are starting to complement and often substitute the static ones in the detection of Android malware. Dynamic techniques analyse the run time behaviour of the malicious code while it is being executed (Egele et al., 2008). For example, Aafer et al. (2013) present *DroidAPIminer*, a new detection system that observes the most frequently used Android API calls, while Shabtai et al. (2010) propose a similar framework called *Andromaly*, that can extract many types of dynamic features.

Furthermore, many works have adopted ML and DL techniques for both static and dynamic malware analysis (David & Netanyahu, 2015). Aonzo et al. (2020) present *BAdDroIds*, a mobile application that leverages deep learning for detecting malware on resource-constrained devices. McLaughlin et al. (2017) propose a deep convolutional neural network to analyse raw sequences of opcodes extracted from disassembled Android malware. Kolosnjaji et al. (2016) use DNNs, convolutional layers, and recurrent layers to analyse the sequence of system calls extracted during apps' execution. D'Angelo et al. (2020) propose a malware detector based on sparse Autoencoders and dynamic sequences of API calls.

Lastly, since Android-based devices are often used in each context, another important research area is the Internet Of Things (IoT) world. IoT devices have often worked with Android OS, and they are capable of collecting and exchanging massive quantities of data at every moment. IoT devices are usually equipped with sensors that measure, detect, and store changes in their environment as data. At the same time, they use the Internet to connect to other devices and systems for sharing their data. However, a recent survey estimated that 20% of companies had experienced at least one IoT-based attack, and IoT's worldwide spending on security will increase from $1.5 billion in 2018 to $3.1 billion in 2021 (Gartner, 2018) to protect themselves from these threats. For example, H. Li et al. (2018) propose a DL schema to reduce network traffic, while Ghosh and Grolinger (2019) propose a DL approach for data dimensions reduction with Autoencoders.

3. Analysed features

As just discussed previously, there are two types of malware analysis approaches, namely, Static Analysis and Dynamic Analysis. In the first one, program-syntax behaviour is analysed by performing reverse engineering operations. The obtained features constitute static information available for describing the program structure and execution flow. In the second one, the run time behaviour of the malicious code is analysed, and its features become dynamic information describing the program runtime activities. Generally, Static and Dynamic techniques are often combined to improve the probabilities of identifying an application as malware and collecting many types of features that bring with them fundamental knowledge about the involved program. For these reasons, and according to related work experiences, we decided to use the following static and dynamic features for describing the program behaviour:

- *Hash Fingerprints:* Hash functions are often used to identify an application as malware. They are based on the concept of collision and a mathematical theory about the difficulty of an attacker to find two arbitrary strings that have the same hash value. It is possible to use many types of hashing algorithms, which are also used to create digital fingerprints during a forensics analysis. For this reason, we decided to include the following hash values: *MD5*, *SHA1*, *SHA256* and *SHA512*.
- *Permissions:* Since the execution of Android applications is based on permission mechanisms, another type of feature that has been included in the proposed dataset is the list of permissions of an application. Generally, these are always contained in an XML file called AndroidManifest, and they are usually classified according to their level of danger. For example, an application could require one or more permissions to execute extra or unnecessary actions.
- *Application Info:* There is other static information that could be used during a malware analysis, such as activities, services, receivers, and providers. Because they usually play an important role during a mobile application's life-cycle, we decided to include in our dataset this extra information for each analysed application.
- *Network Flows:* Android malware applications usually use the network to send potentially sensitive information or install other viruses on the end-user devices. Therefore, some useful features (D'Angelo & Palmieri, 2021) to be used for analysing a mobile application are: Hosts involved, DNS requests, and HTTP/HTTPS requests. This information is usually collected by network analysis tools, such as *Tpcdump*, *TcpFlow*, *Wireshark* and so on. In general, all network flows are stored as PCAP files, which contain all network packets collected during the dynamic analysis. For these reasons, we decided to include in our dataset the list of network requests and two types of PCAP files (i.e. normal and sorted).
- *Dynamic API Calls:* Another operation that is usually done during a malware analysis consists of tracing the behaviour of an application. In particular, it is possible to trace the flow of API calls or System calls. These calls can be collected with specific tools, such as: *Frida*, *Appmon*, *Strace* and *Droidmon* (Frida, 2020; Idanr, 2020b; Patnaik, 2020). However, we cannot use these tools simultaneously on the same process because they try to use another tool, called *Ptrace*. Therefore, we decided to include in our dataset the sequence of API calls invoked during the app's execution instead of analysing only the presence or absence of certain API calls or System calls. Besides, we included a timestamp and other information about the invoked method for each stored API call as a JSON file.

4. CuckooDroid sandbox

In recent years, several Android emulation and runtime analysis solutions, often based on sandboxing technologies, have been proposed, such as for example *Droidbox* (Desnos & Lantz, 2011), *DroidMat* (Wu et al., 2012) and *Copperdroid* (Tam et al., 2015). These solutions supported multiple facilities such as Permissions, Intents, and API calls as the features that can be used for their analysis activity. While quite useful, these solutions are not fully able to tackle anti-emulation. Fortunately, it is possible to use a set of advanced sandboxes, such as Genymotion, based on Mobile Security Framework (MobSF), and CuckooDroid (Abraham, 2020; CuckooDroid, 2020; Genimotion, 2020; Idanr, 2020a), to extract static and dynamic information. For our malware analysis, we decided to use CuckooDroid

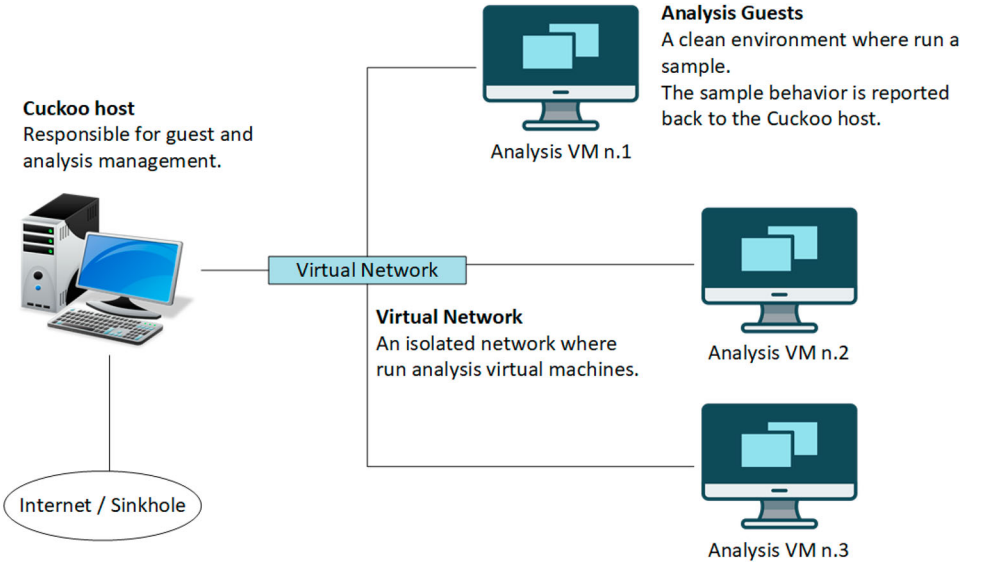

Figure 1. High-level architecture of Cuckoo.

because it is a specific extension of Cuckoo to analyse Android applications. In addition, also CuckooDroid is an open-source tool (Guarnieri et al., 2020a, 2020b).

As just remarked, CuckooDroid is an extension of Cuckoo, and it is an open-source sandbox that was written with Python 2.7. Both present the same high-level architecture, as it is shown in Figure 1. This architecture consists of two main components: a Host part and a Guest part.

Generally, the first one is a Host machine that runs the central management software, while the Guest part consists of one or more physical or virtual machines where the applications are analysed. In particular, the central management software allows the submission of applications to be analysed and permits communication among the host machine and the guest machines. However, the CuckooDroid sandbox can communicate with only one of the guest machines, which can be of three types: *Android On Linux Machine* (with an Android emulator installed on a Linux machine), *Android device cross-platform* (with a physical Android-based device), and *Android emulator* (with an Android emulator installed on the host machine). For our analysis, we used the third type of guest machine (i.e. Android Emulator), which consists of the following modules, as shown in Figure 2:

- *Python Agent:* is a Python script that permits to manage the communication between the host machine and the guest machine.
- *Android Analyser:* is a Python script installed during the configuration of the guest machine.
- *Xposed:* is a framework that can interact with the other modules or to communicate with the other applications.
- *Superuser:* is an Android application that can manage root permissions required by any applications.
- *Content Generator:* is an Android application that can generate a random list of contacts.

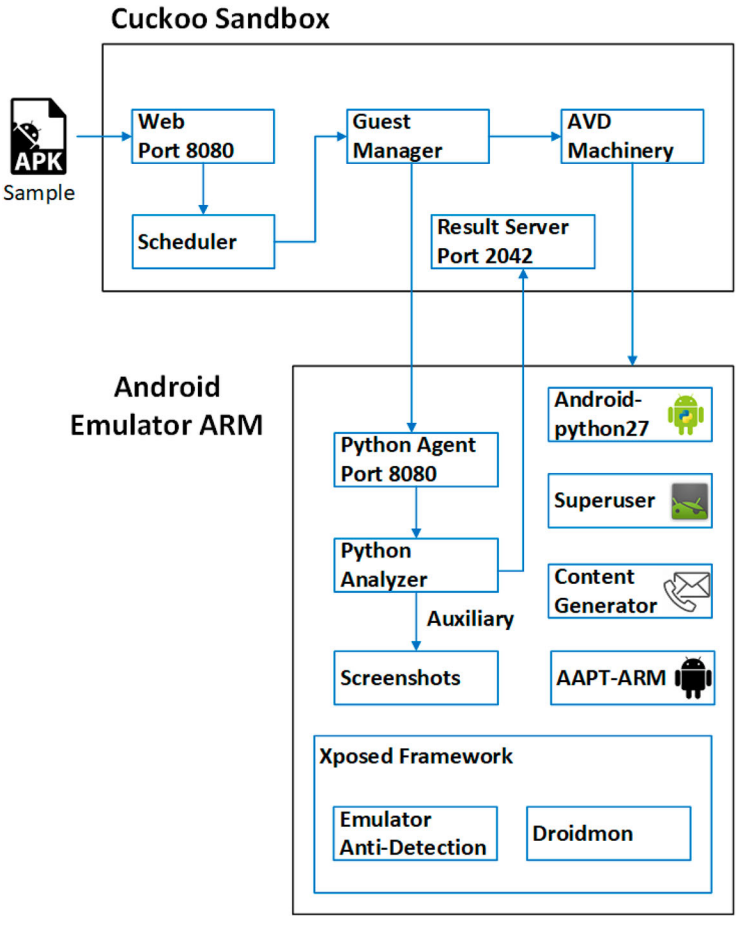

Figure 2. Android emulator architecture.

- *AAPT-ARM:* is a special module that can extract the Main Activity name and other important information from the analysed application.

For each analysed application, CuckooDroid creates a dedicated directory identified with an ID. The final files are stored in such a directory. For example, *analysis.log* is a file that contains the high-level information about the analysis done, *dump.pcap* and *dump_sorted.pcap* contain the network packets that were collected during the analysis done, *report.html* and *report.JSON* contain static information, such as hash values, network requests, activities, services and so on.

4.1. Dynamic API calls

As already remarked, the dynamic API calls play an important role in Android malware analysis because they can show an application's behaviour. According to Section 3, we decided to include in our dataset the sequence of API calls invoked during the app's execution instead of analysing only the presence or absence of specific API calls or System calls. Besides, we

```
{"timestamp":1595723134491,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134521,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134554,"result":"false","class":"java.io.File","method":
{"timestamp":1595723134652,"result":"true","class":"java.io.File","method":"
{"timestamp":1595723134655,"result":"\/data\/data\/com.freegame.kkknfjbgbgra
{"timestamp":1595723134699,"result":"358240051111110","class":"android.telep
{"timestamp":1595723134722,"result":"false","class":"java.io.File","method":
```


Figure 3. Droidmon.log file.

included a timestamp and other information about the invoked method for each stored API call as a JSON file.

The CuckooDroid sandbox can collect dynamic API calls. In particular, it is based on two very important modules: *Xposed* and *Droidmon*, as shown in Figure 2. More precisely, Droidmon is an Android application that can trace API calls and store them in the Logcat file. Each API call is saved as a JSON object in the Logcat file with the help of another file called hooks.JSON, which contains the list of API calls to trace as JSON objects. At the end of each analysis, CuckooDroid saves each dynamic API call (that were previously stored in the Logcat file) in two files: *droidmon.log* and *droidmon_error.log*, as it is shown in Figure 3.

5. The proposed approach

The proposed approach consists of three important steps: (i) analysis of the malware applications and extraction of their features by using CuckooDroid; (ii) representation of their dynamic behaviour (dynamic API calls) with a sequence of sparse matrices, also called API-Images (D'Angelo et al., 2020); and (iii) classification of the analysed applications in multiple families of malware by training a Convolutional Neural Network (CNN) (Lu, 2020; Srivastava & Biswas, 2020) and a Recurrent Neural Network (RNN) (Heinrich & Wermter, 2018), based on the above sparse matrices of features.

5.1. Dynamic behaviour with the API-images

As described in D'Angelo et al. (2020), API-images are a set of sparse matrices representing the dynamic behaviour for each analysed application. In particular, for each Android application, an API-image can be considered as a complete snapshot that represents the dynamic behaviour obtained during the analysis with CuckooDroid. In particular, each API-Image creation consists of two phases: (i) identification of each API call with a unique ID number; and (ii) creation and population of the API-image matrices.

The first step can be conducted by creating a dictionary where each API call has an ID number used to translate the textual information into numerical information. Then, for each application, we consider each two consecutive numerical calls as a pair of coordinates and use them as indexes to store a numeric value in the sparse matrix associated with the considered application. In other words, each consecutive pair of dynamic API calls is used to draw a fixed point in the API-image.

For example, let a1, a2, and a3 be three APIs where a2 is called after a1 and a3 is called after a2. We can consider two pairs of coordinates, $P1 = (1, 2)$ and $P2 = (2, 3)$, respectively, and we can use these pairs to draw two points in the API-image. The pair P1 is associated

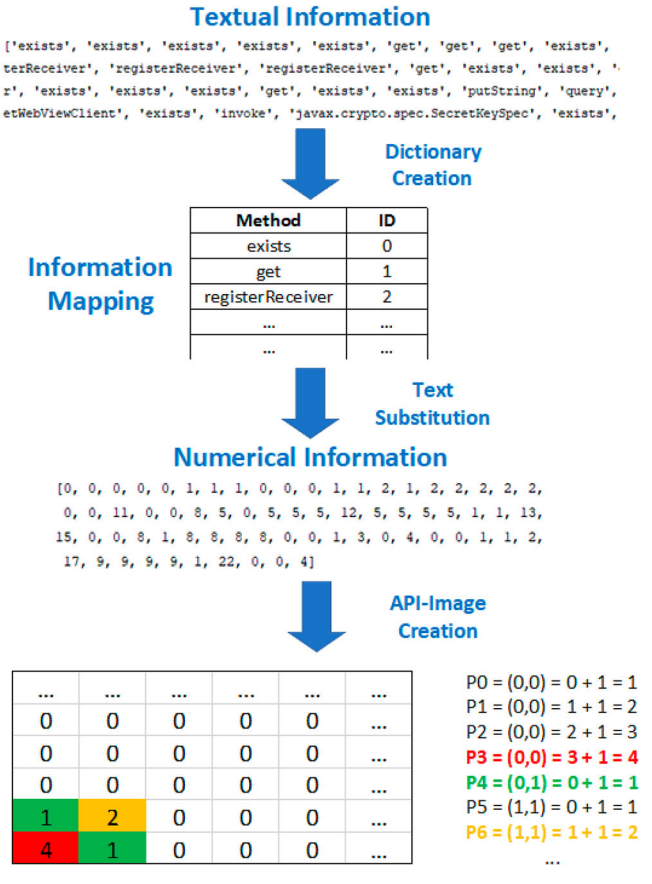

Figure 4. The workflow to obtain an API-image.

with the point identified by the consecutive API calls a1 and a2, while the pair P2 is associated with the point referring to the consecutive API calls a2 and a3. Again, applications may invoke several times, in their activity, the same sequence of API calls, so that some pair can occur several times, that is, more than once. In terms of API-image, this situation can be represented by multiple dots located at the same coordinates, but by using a different colour (RGB or Gray-scale). Figure 4 shows the workflow to obtain an API-image from the proposed dataset, while Figure 5 shows 2 API-images representing the malware families *Airpush* and *Dowgin* respectively. In particular, for each image, a set of 10 malware has been considered, and an RGB image has been used to show the differences among the consecutive API call pairs. Indeed, for each fixed point, a light colour represents two consecutive API calls repeated a few times. In comparison, a dark colour represents two consecutive API calls that are repeated many times. Besides, it is possible to observe both images' sparse behaviour and how it uniquely characterises each malware family.

5.2. The neural networks used for classification

Since API images show the run-time behaviour as pictures, for testing malware classification with our dataset, we used two different kinds of neural networks: Convolutional Neural

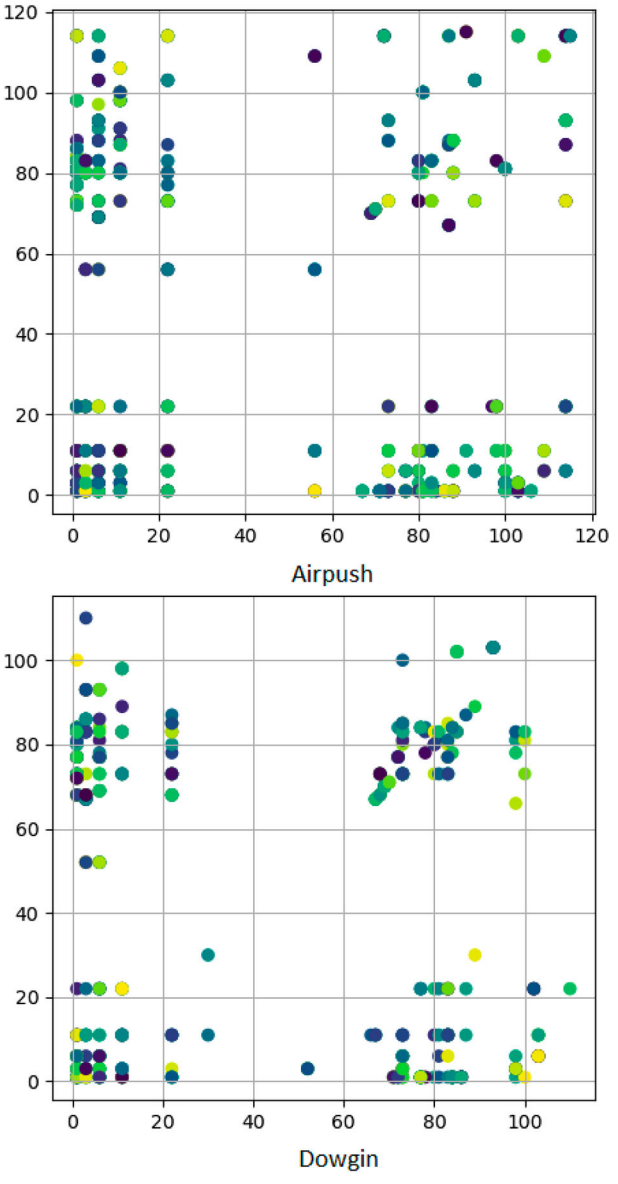

Figure 5. An example of 2 API-images.

Networks (CNNs) and Recurrent Neural Networks (RNNs), which can work effectively on pictures and temporal instances respectively.

CNNs are mostly used in computer vision, and they are often employed to identify images and implement object recognition within scenes. Because the CNNs are a particular class of DNNs, they can process big amounts of data (Big Data) or data that contain much information, such as images or videos. CNN has got different types of input, output, and hidden layers (Kamimura & Takeuchi, 2020; Pan et al., 2020). In particular, the first layers of a CNN can be a set of *Convolutional* and *Pooling* layers, which can reduce and extract the most important features from the input data (e.g. a picture). Another important hidden

layer is the *Flatten* layer, which can prepare the extracted features and use them as an Artificial Neural Network (ANN) input. This operation is often called *flattening*, and it can convert a matrix of features into a one-dimensional vector. Finally, the output part often consists of an ANN used to obtain a classification of the input data (Bhagwat et al., 2019).

On the other hand, RNNs can work on instances structured as progressive observations (i.e. time series) by considering their mutual dependencies and evolution over time. In this way, a given output depends on the previous ones. An RNN could present many recurrent layers, while in classical ANNs (also known as feed-forward neural networks), the data only flows in one direction. In an RNN, the hidden layers create cycles in the network's structure, which are also able to capture recurrence phenomena. In order to build an RNN, we can use different types of input, output, and hidden layers. These usually are SimpleRNN or Long Short Term Memory (LSTM) layers, while the output part often consists of an ANN used to obtain a classification of the input data (Bhagwat et al., 2019).

6. Dataset creation

The proposed dataset was created by analysing two famous malware datasets: *Android Malware Dataset (AMD)* (Y. Li et al., 2017; Wei et al., 2017, june) and *Drebin* (Arp et al., 2014; Spreitzenbarth et al., 2013). In particular, AMD contains 24.553 applications organised into 71 families, while Drebin contains 5560 applications organised into 179 families. The Tables 1 and 2 show the five most important families for AMD and Drebin, respectively.

Out of 30,113 applications analysed, only 25,275 of them were considered and renamed with their SHA256 value. The remaining ones were found to be damaged apps. We collected 20426 malware organised into 66 families for AMD and 4849 malware organised into 143 families for Drebin, respectively. The Tables 3 and 4 show the five most important families resulting after the analyses on the AMD and Drebin datasets, respectively.

Table 1. Five most important families of AMD.

Family	Airpush	Dowgin	FakeInst	Mecor	Youmi
Apps	7843	3385	2172	1820	1301

Table 2. Five most important families of Drebin.

Family	FakeInst	D.K.Fu	Plankton	Opfake	GinMaster
Apps	925	667	625	613	339

Table 3. AMD families after the analyses done.

Family	Airpush	Dowgin	FakeInst	Mecor	Youmi
Apps	5989	2985	2167	1820	1244

Table 4. Drebin families after the analyses done.

Family	FakeInst	D.K.Fu	Opfake	Plankton	BaseBridge
Apps	881	635	607	443	314

Table 5. UMD dataset details summary.

	Total Apk	Analysed Apk	Families	Dim. (Gb)
AMD	24553	20426	66	100.08
Drebin	5560	4849	143	17.55
Total	30113	25275	209	117.63

6.1. Data organisation

We organised our dataset in two main directories: *amd-cuckoo-family* and *drebin-cuckoo-family* which contain 66 and 143 families, respectively. For each of them, the applications are saved using their reports, and the respective SHA256 values identify them. More precisely, for each application, the following files have been stored:

- *report.json:* is a JSON file containing the results of the static analysis like hash values, permissions, intents, services, and so on.
- *dump.pcap:* is a PCAP file containing the network packets collected during the dynamic analysis.
- *dump_sorted:* is a sorted PCAP file containing the collected network packets captured during the dynamic analysis.
- *droidmon.log:* is a textual file containing the dynamic APIs calls stored as JSON objects.
- *droidmon_error.log:* is a textual file containing the dynamic APIs calls that threw a Java Exception.

Lastly, the dimensions of both main folders have been calculated. Table 5 shows a summary of the information just discussed.

7. Experimental results

The dataset used during the experimental evaluation includes 5 different malware families: *Airpush*, *Dowgin*, *FakeInst*, *DroidKungFu (D.K.Fu)*, and *Opfake*, which have been selected in accordance with the Tables 3 and 4. In particular, FakeInst has been chosen because it is a common family between two datasets and is the most representative of Drebin's family; Airpush and Dowgin have been selected because they are the first two representative families of AMD; D.K.Fu and Plankton have been selected because they are the second and the third representative families of Drebin. Subsequently, for each family, we considered 500 samples. Then, for each analysed application, we extracted its dynamic API calls and represented them by using a 116×116-sized API-image, in accordance with the maximum number of different consecutive API calls that have been observed.

Then, we built two neural networks to evaluate the proposed ML-based malware classification framework's performances based on the UMD dataset. In particular, a CNN and an RNN have been used because, as it is remarked in Subsection 5.2, they are able to work effectively on pictures and temporal instances, respectively. The neural networks have been trained with 150 epochs, batch-size 256, and *Adam* optimisation algorithm. *Conv2D*, *Max Pool*, *Flatten*, and *Dense* layers have been used for the CNN; while *SimpleRNN* layers and

Table 6. CNN performance metrics.

	Acc.	Spec.	Prec.	Sens.	F-Score
D.K.Fu	0.9960	0.9948	0.9814	1.0000	0.9906
FakeInst	0.9987	1.0000	1.0000	0.9928	0.9964
Opfake	0.9972	1.0000	1.0000	0.9869	0.9935
Airpush	1.0000	1.0000	1.0000	1.0000	1.0000
Dowgin	1.0000	1.0000	1.0000	1.0000	1.0000

Table 7. CNN average metrics.

	Acc.	Spec.	Prec.	Sens.	F-Score	AUC
Avg.	0.9984	0.9990	0.9963	0.9960	0.9961	1.0000
Dev.	0.0016	0.0023	0.0082	0.0058	0.0040	0.0000

Table 8. RNN performance metrics.

	Acc.	Spec.	Prec.	Sens.	F-Score
D.K.Fu	0.9987	1.0000	1.0000	0.9927	0.9964
FakeInst	1.0000	1.0000	1.0000	1.0000	1.0000
Opfake	1.0000	1.0000	1.0000	1.0000	1.0000
Airpush	1.0000	1.0000	1.0000	1.0000	1.0000
Dowgin	0.9987	0.9982	0.9935	1.0000	0.9966

Table 9. RNN average metrics.

	Acc.	Spec.	Prec.	Sens.	F-Score	AUC
Avg.	0.9995	0.9997	0.9987	0.9986	0.9986	0.9993
Dev.	0.0006	0.0006	0.0028	0.0031	0.0019	0.0007

Dense layers have been used for the RNN. We evaluated each neural network with performance metrics that have been derived by the multi-class confusion matrix, as they are shown in the Tables 6, 7, 8, and 9.

In particular, to assess the performance of the classifiers implemented with the aforementioned neural networks, we used the k-Fold cross-validation algorithm (with $k = 10$) and the 70/30 criteria to split our experimental dataset into two mutually exclusive sets (training set and test set). Moreover, we considered the following classification metrics: Accuracy (Acc.), Sensitivity (Sens.), Specificity (Spec.), Precision (Prec.), Area Under the ROC Curve (AUC), and F-Measure (F-Meas or F-Score). Their values are defined in the same range [0, 1], where 0 represents the worst result, while 1 represents the best result. Subsequently, to obtain a global validation, the average values (Avg.) and standard deviation values (Dev.) among all metrics have been calculated. The averages of the obtained results are shown in the Tables 6 and 8. Here, for each family, TP (true positive) refers to malware instances correctly identified in the considered family, TN (true negative) are the malware instances in another family correctly identified, FP (false positive) considers malware instances incorrectly identified as malware in the considered family, and FN (false negative) refers to malware instances in another family incorrectly recognised as a malware belonging to the considered family.

Table 10. A comparison between the CNN and RNN-based methods and the more traditional ones.

	Acc.	Spec.	Prec.	Sens.	F-Score	AUC
CNN	0.9984	0.9990	0.9963	0.9960	0.9961	1.0000
RNN	0.9995	0.9997	0.9987	0.9986	0.9986	0.9993
MLP	0.9990	0.9950	0.9950	0.9950	0.9950	1.0000
J48	0.9970	0.9870	0.9870	0.9870	0.9870	0.9960
NB	0.9710	0.8970	0.8890	0.8870	0.8860	0.9900

Finally, a comparison with the most notable approaches available in literature was made using WEKA (WEKA, 2020). In particular, we used 3 famous classifier technologies: *Multi-Layer Perceptron (MLP)*, *J48 trees (J48)*, and *Naive Bayes (NB)*. The obtained results are shown in Table 10. It is possible to see how these results are excellent for all classification methods, except for Naive Bayes. This aspect is due to the effectiveness of API-images in capturing the malware dynamic runtime behaviour. However, the CNN- and RNN-based models achieve the best solutions in recognising the different malware families because the obtained values (in particular Precision, Sensibility, and F-Score) are higher than those obtained with the other classification methods. Therefore, both classification models can reduce the number of FP and FN and, consequently, better minimise the classification error.

8. Conclusions and future works

In this paper, the realisation of a new Android malware dataset called Unisa Malware Dataset (UMD) has been presented. UMD is available on http://antlab.di.unisa.it/malware/. The proposed dataset has been realised by analysing 30,113 malware applications through CuckooDroid Sandbox. More precisely, the UMD contains 20,426 apps organised into 66 families for AMD and 4849 applications organised into 143 families for Drebin, respectively. Besides, for each analysed application, static and dynamic features are available, such as hash fingerprints, permissions, dynamic API calls, and so on. Then, an experiment with Artificial Neural Networks (ANNs) has been presented to show the potentialities of the extracted API calls by considering 5 malware families (Airpush, Dowgin, FakeInst, DroidKungFu, and Opfake). However, UMD is an unbalanced dataset consisting of many malware families with a low number of applications. At the same time, a great number of malware families should be included in our dataset. Consequently, only a limited subset of families can be taken into account in order to propose new AI-based solutions. Furthermore, 500 samples have been selected for each family, and dynamic API calls have been extracted from them as an API-image. Finally, a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) have been used, which have been validated by using statistic metrics. The obtained results show that these neural networks are an effective solution to recognise malware families when the right features are used to describe the behavioural properties of individual malware flavours.

For this reason, we would like to propose two possible future works. First of all, in order to improve the number of malware applications and the number of the considered families, we will update the proposed dataset by considering other malware datasets, such as: *Android Adware and General Malware Dataset* (*AAGM Dataset*) (Habibi Lashkari et al., 2017, august), *AndroZoo* (Allix et al., 2016), *Genome* (Zhou & Jiang, 2012) and so on.

We did not include these datasets yet, because the analysis' process is costly and time-consuming. Second, we will propose new AI models based on the extracted static and dynamic features to improve the already achieved results. For example, a Recurrent Neural Network (RNN) based on Long Short-Term Memory (LSTM) layers could be a good method for use temporal features, such as timestamps. Moreover, the use of CNN autoencoders could be investigated to obtain important features by API-image based on the extracted static and dynamic information. Additionally, we will explore new DL approaches that will able to classify dynamic features as a film. More precisely, several combinations among LSTM layers, CNNs, and Stacked Autoencoders (SAEs) could be investigated to consider a single API-Image as a stream of sub-API-images by taking into account many sets of images obtained at fixed multiple temporal windows.

Acknowledgments

We would offer our sincerest gratitude to two Master's Degree students of our department: Giacomo Cocoziello and Aniello Giugliano, for their help during the experimental phase.

Disclosure statement

No potential conflict of interest was reported by the author(s).

References

Aafer, Y., Du, W., & Yin, H. (2013). DroidAPIMiner: Mining API-level features for robust malware detection in android. In *Securecomm*.

Abraham, A. (2020). *Mobile security framework (MOBSF)*. Retrieved 2020, from https://www.genymotion.com/

Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2016). AndroZoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th international conference on mining software repositories* (pp. 468–471). ACM. http://doi.acm.org/10.1145/2901739.2903508

Aonzo, S., Merlo, A., Migliardi, M., Oneto, L., & Palmieri, F. (2020). Low-resource footprint, data-driven malware detection on android. *IEEE Transactions on Sustainable Computing*, *5*(2), 213–222. https://doi.org/10.1109/TSUSC

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., & Rieck, K. (2014, February). *DREBIN: Effective and explainable detection of android malware in your pocket*.

Bhagwat, R., Abdolahnejad, M., & Moocarme, M. (2019). *Applied deep learning with keras: solve complex real-life problems with the simplicity of keras*. Packt Publishing.

Cerwall, P. (2015). *Ericsson mobility report*. https://www.ericsson.com/assets/local/mobility-report/documents/2015/ericsson-mobility-report-june-2015.pdf

Chan, P. P. K., & Song, W.-K. (2014). Static detection of Android malware by using permissions and API calls. In *2014 International Conference on Machine Learning and Cybernetics*, IEEE (Vol. 1, pp. 82–87).

CuckooDroid (2020). *Cuckoodroid book*. Retrieved 2020, from https://cuckoo-droid.readthedocs.io/en/latest/

D'Angelo, G., Ficco, M., & Palmieri, F. (2020). Malware detection in mobile environments based on autoencoders and API-images. *Journal of Parallel and Distributed Computing*, *137*, 26–33. https://doi.org/10.1016/j.jpdc.2019.11.001

D'Angelo, G., & Palmieri, F. (2021). Network traffic classification using deep convolutional recurrent autoencoder neural networks for spatial-temporal features extraction. *Journal of Network and Computer Applications*, *173*, Article ID 102890. https://doi.org/10.1016/j.jnca.2020.102890

David, O., & Netanyahu, N. S. (2015). DeepSign: Deep learning for automatic malware signature generation and classification. In *2015 International joint conference on neural networks (IJCNN)*, IEEE (pp. 1–8).

Desnos, A., & Lantz, P. (2011). *Droidbox: An android application sandbox for dynamic analysis* [Master's thesis, SwedenLund University].

Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008, March). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, *44*(2), 1–42. https://doi.org/10.1145/2089125.2089126

Ficco, M. (2019, June). Detecting IoT malware by Markov Chain behavioral models. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE (pp. 229–234).

Frida (2020). *Frida – a world-class dynamic instrumentation framework*. Retrieved 2020, from https://frida.re/docs/frida-trace/

Gartner (2018). *Gartner says worldwide iot security spending will reach $1.5 billion in 2018*. https://www.gartner.com/en/newsroom/press-releases/2018-03-21-gartner-says-worldwide-iot-security-spending-will-reach-1-point-5-billion-in-2018

Genimotion (2020). *Genimotion android emulator*. Retrieved 2020, from https://www.genymotion.com/

Ghosh, A. M., & Grolinger, K. (2019, May). *Deep learning: Edge-cloud data analytics for IoT*.

Guarnieri, C., Tanasi, A., Bremer, J., Schloesser, M., Houtman, K., R. V. Zutphen, & Graaff, B. D. (2020a). *Cuckoo*. Retrieved 2020, from https://github.com/cuckoosandbox/cuckoo

Guarnieri, C., Tanasi, A., Bremer, J., Schloesser, M., Houtman, K., Zutphen, R. V., & Graaff, B. D. (2020b). *Cuckoo sandbox – automated malware analysis*. Retrieved 2020, from https://cuckoosandbox.org/

Habibi Lashkari, A., Abdul kadir, A. F., Gonzalez, H., Mbah, K., & Ghorbani, A. (2017, August). Towards a network-based framework for android malware detection and characterization. In *2017 15th Annual conference on privacy, security and trust (PST)*, IEEE (pp. 233–23309).

Heinrich, S., & Wermter, S. (2018). Interactive natural language acquisition in a multi-modal recurrent neural architecture. *Connection Science*, *30*(1), 99–133. https://doi.org/10.1080/09540091.2017.1318357

Idanr (2020a). *Cuckoodroid – automated android malware analysis*. Retrieved 2020, from https://github.com/idanr
1986/cuckoo-droid

Idanr (2020b). *Droidmon – dalvik monitoring framework for cuckoodroid*. https://github.com/idanr1986/droidmon

Kamimura, R., & Takeuchi, H. (2020). Improving collective interpretation by extended potentiality assimilation for multi-layered neural networks. *Connection Science*, *32*(2), 174–203. https://doi.org/10.1080/09540091.2019.1674245

Karbab, E. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, *24*, S48–S59. https://doi.org/10.1016/j.diin.2018.01.007

Kolosnjaji, B., Zarras, A., Webster, G., & Eckert, C. (2016). Deep learning for classification of malware system call sequences. In B. H. Kang & Q. Bai (Eds.), *AI 2016: Advances in artificial intelligence* (pp. 137–149). Springer International Publishing.

Lashkari, A. H., Kadir, A. F. A., Taheri, L., & Ghorbani, A. A. (2018). Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 International Carnahan Conference on Security Technology (ICCST)*, IEEE (pp. 1–7).

Li, Y., Jang, J., Hu, X., & Ou, X. (2017). Android malware clustering through malicious payload mining. In *Lecture Notes in Computer Science*, Springer (pp. 192–214).

Li, H., Ota, K., & Dong, M. (2018). Learning IoT in edge: Deep learning for the internet of things with edge computing. *IEEE Network*, *32*(1), 96–101. https://doi.org/10.1109/MNET.2018.1700202

Lu, T. C. (2020). CNN convolutional layer optimisation based on quantum evolutionary algorithm. *Connection Science*. https://doi.org/10.1080/09540091.2020.1841111

Martín García, A., Rodriguez-Fernandez, V., & Camacho, D. (2018, June). CANDYMAN: Classifying android malware families by modelling dynamic traces with Markov chains. *Engineering Applications of Artificial Intelligence*, *74*, 121–133. https://doi.org/10.1016/j.engappai.2018.06.006

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., & Joon Ahn, G. (2017). Deep android malware detection. In *Proceedings of the seventh ACM on conference on data and application security and privacy* (pp. 301–308). Association for Computing Machinery. https://doi.org/10.1145/3029806.3029823

Ogiela, L. (2010). Cognitive informatics in automatic pattern understanding and cognitive information systems. In *Advances in cognitive informatics and cognitive computing* (pp. 209–226). Springer.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., & Stringhini, G. (2019, April). MaMaDroid: Detecting android malware by building Markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security*, *22*(2), 1–34. https://doi.org/10.1145/3313391

Pan, L., Li, C., Zhou, Y., Chen, R., & Xiong, B. (2020). A combinational convolutional neural network of double subnets for food-ingredient recognition. *International Journal of Embedded Systems*, *13*(4), 439–448. https://doi.org/10.1504/IJES.2020.110658

Patnaik, N. D. (2020). *Welcome to appmon!* Retrieved 2020, from https://github.com/dpnishant/appmon

Potha, N., Kouliaridis, V., & Kambourakis, G. (2020). An extrinsic random-based ensemble approach for android malware detection. *Connection Science*. https://doi.org/10.1080/09540091.2020.1853056

Reina, A., Fattori, A., & Cavallaro, L. (2013, April 14). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *ACM European workshop on systems security (EUROSEC)*. ACM.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2010). "Andromaly": A behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, *38*, 161–190. https://doi.org/10.1007/s10844-010-0148-x

Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., & Hoffmann, J. (2013). Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th annual ACM symposium on applied computing* (pp. 1808–1815). Association for Computing Machinery. https://doi.org/10.1145/2480362.2480701

Srivastava, V., & Biswas, B. (2020). CNN-based salient features in HSI image semantic target prediction. *Connection Science*, *32*(2), 113–131. https://doi.org/10.1080/09540091.2019.1650330

Statista (2019). *Annual number of global mobile app downloads 2016–2021, by store*. https://www.statista.com/statistics/276602/annual-number-of-mobile-app-downloads-by-store/

Symantec (2018). *Connect symantec archives*. http://www.symantec.com/connect/blogs/yet-another-bunchmalicious-apps-found-google-play

Taheri, L., Kadir, A. F. A., & Lashkari, A. H. (2019). Extensible android malware detection and family classification using network-flows and API-calls. In *2019 International Carnahan Conference on Security Technology (ICCST)*, IEEE (pp. 1–8).

Tam, K., Khan, S. J., Fattori, A., & Cavallaro, L. (2015). *Copperdroid: Automatic reconstruction of android malware behaviors*. https://core.ac.uk/download/pdf/77298524.pdf

Tian, Q., Han, D., Li, K. C., Liu, X., Duan, L., & Castiglione, A. (2020). An intrusion detection approach based on improved deep belief network. *Applied Intelligence*, *50*(10), 3162–3178. https://doi.org/10.1007/s10489-020-01694-4

Wei, F., Ou, X., & Zhou, W. (2017, June). Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, Cham (pp. 252–276).

WEKA (2020). *Weka 3 – data mining with open source machine learning software in java*. Retrieved 2020, from https://www.cs.waikato.ac.nz/ml/weka/

Wu, D., Mao, C., Wei, T., Lee, H., & Wu, K. (2012). DroidMat: Android malware detection through manifest and API calls tracing. In *2012 Seventh Asia Joint Conference on Information Security, IEEE* (pp. 62–69).

Zhang, M., Duan, Y., Yin, H., & Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 1105–1116). Association for Computing Machinery. https://doi.org/10.1145/2660267.2660359

Zhou, Y., & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, IEEE (pp. 95–109).