10-2020

Experimental comparison of features and classifiers for Android malware detection

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Biniam Fisseha DEMISSIE
Fondazione Bruno Kessler

Mariano CECCATO
University of Verona

Wei MINN
Singapore Management University, wei.minn.2018@sis.smu.edu.sg

Citation

SHAR, Lwin Khin; DEMISSIE, Biniam Fisseha; CECCATO, Mariano; and MINN, Wei. Experimental comparison of features and classifiers for Android malware detection. (2020). *MOBILESoft 2020: Proceedings of the 7th IEEE/ACM International Conference on Mobile Software Engineering and Systems, Seoul, South Korea, October 5-6*. 50-60.
Available at: https://ink.library.smu.edu.sg/sis_research/5115

Experimental Comparison of Features and Classifiers for Android Malware Detection

Lwin Khin Shar
Singapore Management University
lkshar@smu.edu.sg

Biniam Fisseha Demissie
Fondazione Bruno Kessler
demissie@fbk.eu

Mariano Ceccato
University of Verona
mariano.ceccato@univr.it

Wei Minn
Singapore Management University
wei.minn.2018@sis.smu.edu.sg

ABSTRACT

Android platform has dominated the smart phone market for years now and, consequently, gained a lot of attention from attackers. Malicious apps (malware) pose a serious threat to the security and privacy of Android smart phone users. Available approaches to detect mobile malware based on machine learning rely on features extracted with static analysis or dynamic analysis techniques. Different types of machine learning classifiers (such as support vector machine and random forest) deep learning classifiers (based on deep neural networks) are then trained on extracted features, to produce models that can be used to detect mobile malware. The usually-analyzed features include permissions requested/used, frequency of API calls, use of API calls, and sequence of API calls. The API calls are analyzed at various granularity levels such as method, class, package, and family.

In the view of the proposals of different types of classifiers and the use of different types of features and different underlying analyses used for feature extraction, there is a need for a comprehensive evaluation on the effectiveness of the current state-of-the-art studies in malware detection on a common benchmark. In this work, we provide a baseline comparison of several conventional machine learning classifiers and deep learning classifiers, without fine tuning. We also provide the evaluation of different types of features that characterize the use of API calls at class level and the sequence of API calls at method level. Features have been extracted from a common benchmark of 4572 benign samples and 2399 malware samples, using both static analysis and dynamic analysis.

Among other interesting findings, we observed that classifiers trained on the use of API calls generally perform better than those trained on the sequence of API calls. Classifiers trained on static analysis-based features perform better than those trained on dynamic analysis-based features. Deep learning classifiers, despite their sophistication, are not necessarily better than conventional classifiers, especially when they are not optimized. However, deep learning classifiers do perform better than conventional classifiers when trained on dynamic analysis-based features.

KEYWORDS

Malware detection, machine learning, deep learning, Android

ACM Reference Format:
Lwin Khin Shar, Biniam Fisseha Demissie, Mariano Ceccato, and Wei Minn. 2020. Experimental Comparison of Features and Classifiers for Android Malware Detection. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387905.3388596

1 INTRODUCTION

Android platform has dominated the smart phone market for years now. With currently more than two billion devices running Android, it is the most popular end-user operating system in the world. Its market dominance and open source nature has also made it interesting for attackers. Symantec [40] reported that in 2018, it detected an average of 10573 mobile malware per day; found that one in 36 mobile devices had high risk apps installed; and one in 14.5 apps accesses high risk user data. Hence, Android malware detection is currently an active area of research.

There have been a number of approaches proposed by the research community to detect Android malware. Many approaches have built malware detection models based on permissions request/use [5, 9, 16, 20, 26, 34, 36].

However, since benign apps also often request permissions classified as dangerous for legitimate reasons, permission-based approaches can be prone to false positives [16]. More recent approaches have built detection models based on sequence of API calls [23, 30, 41], use of API calls [5, 9, 36, 47] or frequency of API calls [1, 19].

The API calls can be extracted at various granularity levels such as method, class, package, and family. Since there are millions of unique methods in Android, some approaches [19, 21, 30] that are based on the use or the frequency of API calls have proposed to abstract API calls at class, package, and/or family levels. This reduced the number of features significantly and yet produced comparable or even better results [19, 21, 30].

To extract these features, in general two types of techniques are used — static analysis [5, 9, 19, 21, 30, 46] and dynamic analysis [15, 41]. For instance, Drebin [5] extracts permissions and API calls by scanning manifest files and disassembled code. DadiDroid [21] and

MOBILESoft '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7959-5/20/05…$15.00
https://doi.org/10.1145/3387905.3388596

MamaDroid [30] extract API calls from call graphs. The majority of the approaches has relied on static analysis for feature extraction. Those approaches that apply dynamic analysis [15, 41] have mainly focused on features at native level API calls (system calls). Typically, static analysis-based features cover more information since static analysis can reason with the whole program code whereas dynamic analysis-based features are limited to the code that are executed. On the other hand, static analysis may have issues dealing with complex code such as code obfuscation, and modern malware are usually crafted with obfuscated code [19]. Hence, in general, static analysis and dynamic analysis complement each other.

Once these features are extracted using program analyses, these approaches typically use machine learning classifiers to train on the features and build malware detection model. For instance, Support Vector Machines (SVM), K-Nearest Neighbours, and Random Forest were used in [21, 30]; AdaBoost, Naive Bayes, Decision Tree, and SVM were used in [20].

In parallel, other studies [23, 27, 41, 45] have focused on the use of deep learning classifiers, such as Convolutional Neural network and Recurrent Neural Network, instead of conventional machine learning classifiers, to build malware detectors. Deep learning classifiers use several layers to study various levels of representations and extract higher-level features from the given lower-level ones. Hence, in general they have built-in feature selection process and are better at learning complex patterns.

In the view of the proposals of different types of classifiers and the use of different types of features and different analyses used for feature extraction, there is a need for a comprehensive evaluation on the effectiveness of the current state-of-the-art in malware detection on a common benchmark. This study aims to evaluate the malware detection accuracy of various classifiers without fine tuning, when learnt on different types of features from a common benchmark, using both static and dynamic program analyses.

We use 4572 benign samples and 2399 malware samples. Benign samples were randomly collected from Androzoo repository [2], which are released from year 2017 to 2019. 1208 malware samples are collected from Androzoo repository [2], which are from year 2017 and 2019, and 1191 malware samples are from Drebin repository [5]. We extract static features from call graph of Android package (apk) codes and dynamic features by executing the app in an Android emulator using our in-house intent-fuzzer combined with Android's Monkey testing framework [4].

Specifically, we make the following contributions in this paper.

- We evaluate several conventional machine learning classifiers and deep learning classifiers. More specifically, we assess seven machine learning classifiers, namely K-Nearest Neighbors (KNN), Support Vector Machines (SVM), Decision Tree (DT), Random Forest (RF), AdaBoost (AB), Naive Bayes (NB), and Logistic regression (LR). We assess four deep learning classifiers, namely Simple Artificial Neural Network (sANN), Complex Artificial Neural Network (cANN), Convolutional Neural Network (CNN), and Recurrent Neural Network with long short term memory (RNN);
- We compare the malware detection accuracy of using the features that characterize the sequence of API calls and that of using the features that characterize the use of API calls;
- We compare the malware detection accuracy of using the static analysis-based features, the dynamic analysis-based features, and the combined set of static and dynamic analysis-based features. To the best of our knowledge, we have not observed a hybrid approach that utilizes both analyses to extract features on API calls at method and class levels;
- We compare the cost in terms of training time requirement of using different types of features.

We make the dataset and the scripts used in our experiments available [35] so that researchers could replicate or extend our experiments.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 thoroughly discuses the methodology and an overview of malware detection; it explains the data collection and features extraction processes, and the machine learning and deep learning classifiers we use. Section 4 presents the evaluation studies and discusses the experimental results. Section 5 provides the concluding remarks and proposals for future studies.

2 RELATED WORK

Naway and Li [29] reviewed the use of deep learning in combination with program analysis for Android malware detection. However, their contribution was a literature survey (focusing on the differences between key concepts of different DL classifiers and different feature extraction techniques) rather than an empirical study like ours. Experimental comparisons are available in literature, contrasting different types of features and classifiers to detect Android malware. However, these approaches usually compare their single proposed method against other recent approaches. Conversely, our study aims at comparing different types of features and classifiers that have been used by the research community, on a common benchmark.

Static analysis-based features. Several approaches rely on static analysis to extract features from the app such as requested permissions [5, 9, 16, 20, 26, 34, 36], the sequence of API calls [11, 23, 27, 30, 37], the use of API calls [5, 9, 21, 36, 47, 50], or the frequency of API calls [1, 11, 18, 19]. Our study also investigates and compares the performance of using API use features and API sequence features. However, our study is not limited to features extracted with static analysis, but also with dynamic analysis. Additionally, we also study the performance of combining the features obtained from the two analyses, and we evaluate these features across several classifiers.

Like our study, some approaches [21, 30] extract static features from call graphs. Other approaches rely, instead, on data dependency graphs [37, 50] or control flow graph [12]. In future, we plan to evaluate the difference between using different kinds of graphs.

Considering that analysis at method level led to millions of features, resulting in long training time and memory consumption, some approaches [21, 30, 46] abstracted features at class, package, family, or entity levels, to save memory and time. Our study adopts both views, by evaluating features either at method level and at the class level.

Dynamic analysis-based features. Dynamic analysis-based approaches such as [15, 41] have mainly focused on features at native level API calls (system calls). Narudin et al. [28] evaluated the performance of five ML classifiers on network features (API calls that involve network communication) extracted with dynamic analysis. In contrast to these approaches, we consider all the standard APIs, and we evaluate both ML and DL classifiers.

Hybrid analysis-based features. Few approaches [3, 25, 48] apply both static analysis and dynamic analysis techniques. However, these approaches focused on extracting specific features that are generally considered to be dangerous, such as sending SMS and connecting to Internet. By contrast, we do not discriminate features, and we consider a more complete set of features, which are not considered in those approaches. This implies that our test generator has to be more comprehensive to cover more program behaviours. While most dynamic analysis approaches have largely used Monkey (UI) test generator [29], our approach employs a combination of Monkey test generator and intent fuzzing to also cover component interactions.

Deep learning vs Machine learning. Recently, deep learning for Android malware detection has been endorsed [23, 27, 43, 45]. Droidsec [48] compared a deep belief network classifier against conventional ML classifiers such as NB, SVM, and LR. But their study excludes Random Forest, and their dataset was limited to only 250 malware and 250 benign samples. Their results showed that DL classifier is more accurate. On the other hand, MaMaDroid [30] found that conventional ML classifiers like Random Forest and K-Nearest Neighbours perform better.

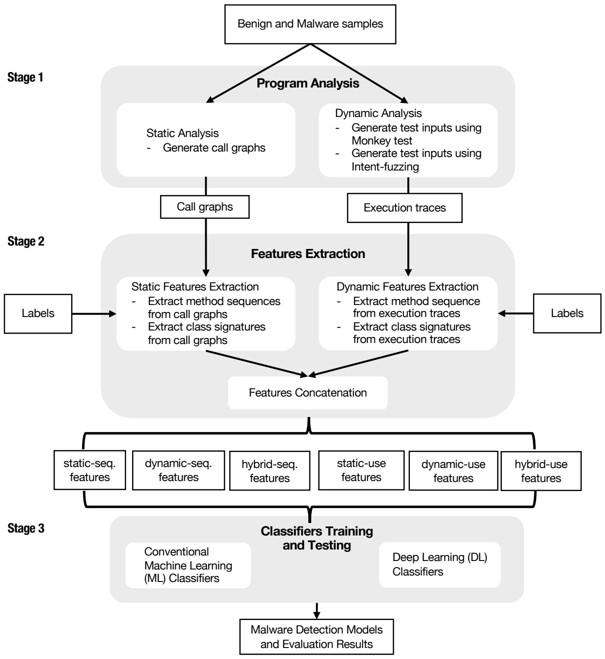

Benign and Malware samples
Stage 1
Program Analysis
Static Analysis
- Generate call graphs
Dynamic Analysis
- Generate test inputs using Monkey test
- Generate test inputs using Intent-fuzzing
Call graphs
Execution traces
Stage 2
Features Extraction
Labels
Static Features Extraction
- Extract method sequences from call graphs
- Extract class signatures from call graphs
Dynamic Features Extraction
- Extract method sequence from execution traces
- Extract class signatures from execution traces
Labels
Features Concatenation
static-seq. features
dynamic-seq. features
hybrid-seq. features
static-use features
dynamic-use features
hybrid-use features
Stage 3
Classifiers Training and Testing
Conventional Machine Learning (ML) Classifiers
Deep Learning (DL) Classifiers
Malware Detection Models and Evaluation Results

Figure 1: The workflow of the experiments

3 METHODOLOGY

This section presents the overall workflow of the experiments. Figure 1 illustrates the workflow, which consists of three stages. The first stage is program analysis, which extracts call graphs and execution traces from benign and malware samples. The second stage is feature extraction in which six datasets are extracted, namely static method sequence features, dynamic method sequence features, hybrid method sequence features, static class features, dynamic class features, and hybrid class features, from call graphs and execution traces. And they are labeled. In the last stage, conventional machine learning classifiers (denoted as *ML classifiers*) and deep learning classifiers (denoted as *DL classifiers*) are trained and tested on the labeled dataset and produce the evaluation results. The following subsections discuss each stage in detail.

3.1 Program Analysis

In this phase, we perform static and dynamic analysis on the given Android application packages (apks).

For static analysis, we use FlowDroid [6] with its default settings, to extract call graphs from an apk. FlowDroid is based on Soot [38]. Firstly, given an apk, Soot converts it into an intermediate representation called Jimple and FlowDroid performs flow analysis on the Jimple code. The analysis is flow- and context-sensitive. FlowDroid also has an optional feature for handling reflections. We opted to use this feature since Android malware increasingly makes use of reflection to avoid detection. However, like other static analysis tools, FlowDroid also shares inherent limitation of static analysis. It can only resolve reflective calls when the arguments used in the call are all string constants. Dynamic analysis can overcome this limitation. In addition, FlowDroid also handles common native calls, i.e., using some heuristics, it can track data flow across some commonly used native calls.

Dynamic analysis is performed in two phases: the first phase analyzes call graph of the app to extract paths from public entry-points (i.e., inter-component communication interfaces) to the leaf nodes. Similar to the static analysis phase, we generate the call graph of the app using Soot with FlowDroid plugin for Android. The call graph is then traversed forward in depth-first search manner starting from the root node until a leaf node is reached. The output of this step is paths from the roots (entry-points of each component) to the different leaf nodes (method calls without outgoing edges).

Once the list of paths is available, the next step is to instantiate an inter-component communication message (intent) fuzzer to generate inputs that execute the paths. To this end, we first instrument the app to collect method execution traces and install the app on an Android emulator. We then run our intent fuzzer with statically collected values (such as static strings) from the app as seed (initial values).

The generated inputs are Intent messages that are sent to the app under test via the Android Debug Bridge (ADB). Our goal is to maximize coverage and collect as many traces as possible. The traces are also used to guide the test generation.

While this step exercises code parts that involve inter-component (inter-app) communications, it does not address user interactions such as UI inputs.

In order to complement the first phase, we instantiate the second phase that uses Google's Android Monkey tool [4]. Monkey comes with the Android SDK and is used to randomly generate input events such as tap, input text or toggle WIFI in an attempt to trigger abnormal app behaviors.

The combined test generator covers app behaviors in a more comprehensive way. While Monkey tool covers GUI-related features, our fuzzer focuses on exercising inter-component (inter-app) interactions.

3.2 Features Extraction

From the call graphs and the execution traces generated in the previous phase, we extract six datasets as explained in the following:

Extracting from the sequences of API calls. Three datasets are extracted from the sequences of API calls at method level — one from call graphs, one from execution traces, and one from combining the former two. Given a call graph, we traverse the graph in a depth first search manner and extract methods as we traverse (hence, sequence). If there is a loop, the method is traversed only once. Note that we only extract the methods from Android framework classes, Java classes, and standard `Org` classes (`org.apache`, `org.xml`, etc.). This is because it is common for malware to be obfuscated to circumvent malware detectors. The obfuscation often involves renaming of library and custom (user defined) methods and classes. Hence, a malware detector will not be resilient to obfuscation if it is trained on library and custom methods and classes. A previous study has shown that a simple renaming obfuscation method can prevent popular anti-malware products from detecting the transformed malware samples [33]. Hence, we skipped methods that are not from the above-mentioned standard packages as we traverse the call graph. Similarly, we extract methods from the execution traces. However, since execution traces are already sequences, depth first search is not necessary. An excerpt of an extracted sequence is shown in Figure 2.

Next, we discretized the sequence of method calls we extract above so that it can be processed by machine learning and deep learning classifiers. More precisely, we replace each unique method with an identifier, resulting in a sequence of numbers. We build a dictionary that maps each method call to its identifier. During the testing or deployment phase, we may encounter unknown API calls. To address this, (1) we consider a large dictionary that covers nearly 2.9 millions of unique methods from standard libraries and (2) we replace all unknown API calls with a fixed identifier.

The length of the sequences varies from one app to another. Thus, it is necessary to unify the length of the sequences. Since we have two types of method sequences — from call graphs and from execution traces, we chose two different uniform sequence lengths. Initially, we extracted the whole sequences. We then took the median length of sequences from call graphs as the uniform sequence size, denoted as L_{cg}, for call graph-based method sequence features and took the median length of sequences from execution traces as the uniform sequence, denoted as L_{tr}, for execution traces-based method sequence[1]. If the length of a given sequence is less than L, we pad the sequence with zeros; if the length is longer than L, we trim it to L, from the right. Hence, for each app, we end up with a sequence of numbers which is a feature vector. Each number in the sequence corresponds to the categorical value of a feature. The number of features is the uniform sequence length L.

As a result, we obtain one dataset from call graphs that characterizes the sequence of API calls at method level, denoted as *static-sequence features*. Likewise, we obtain one dataset from execution traces, denoted as *dynamic-sequence features*. We also concatenate the two sets of features into one dataset, denoted as *hybrid-sequence features*. In general, we will denote them as *sequence features*.

Figure 3 shows a sample dataset containing the *sequence features*.

```
android.webkit.WebSettings: void setPluginsEnabled(boolean)
android.webkit.WebView: void setVisibility(int)
android.os.Handler: void <init>()
java.lang.Boolean: java.lang.Boolean valueOf(boolean)
android.webkit.WebView: void loadUrl(java.lang.String)
```

Figure 2: An excerpt of a sequence of API calls from a sample

	seq0	seq1	…	seqn	label
benign1	74921	567	…	84111	0
benign2	12901	4490	…	3923	0
mal1	23712	6812	…	0	1
mal2	23	63011	…	0	1

Figure 3: Example of the sequence of API calls features

	WebView	Picture	SQLiteDatabase	label
benign1	1	0	0	0
benign2	0	0	1	0
mal1	1	1	0	1
mal2	0	1	1	1

Figure 4: Example of the use of API calls features

Extracting from the uses of API calls. Three datasets are extracted from the uses of API calls at class level — one from call graphs, one from execution traces, and one from combining the former two. The rationale for choosing class level features instead of method level features is to reduce the amount of features such as those approaches in [19, 21, 30]. Method level features would result in millions of features and yet the classifiers may not achieve a better accuracy since the feature vectors of the samples would be sparse.[2].

The extraction process is the same for both call graphs and execution traces. We initially build a database that stores unique classes. Again for obfuscation resiliency, we only consider the Android framework, Java, and standard `Org` classes as explained above. We currently maintain 134,558 classes. Given call graphs or execution traces, we scan the files and extract the class signatures (sequence does not matter in this case). Each unique class in our database corresponds to a feature. The value of a feature is 1 if the corresponding class is found in the given call graph or execution trace; otherwise, it is 0.

As a result, we obtain one dataset, from call graphs, which characterizes the use of API calls at class level, denoted as *static-use features*. Likewise, we obtain one dataset from execution traces, denoted as *dynamic-use features*. We also concatenate the two sets of

[1] L_{cg}=85000, L_{tr}=20000

[2] we did a preliminary study on the use of method level and class level features on a randomly selected sample set containing 50 benign and 50 malware samples and observed that the classifiers achieved similar results.

features into one dataset, denoted as *hybrid-use features*. In general, we will denote them as *use features*.

Figure 4 shows a sample dataset containing the *use features*.

3.3 Classifiers

In the last phase, classifiers are trained and tested on each dataset extracted in phase 2. The following briefly describes the classifiers used in our evaluations.

3.3.1 Conventional Machine Learning (ML) Classifiers.

We evaluate seven ML classifiers:

K-Nearest Neighbours, KNN is one of the simplest classification techniques, with less or no prior knowledge of data distribution. The predicted test sample class is set equal to the true class among the nearest training instances [24]. In our experiments, three neighbours comprised the KNN setting to perform the classifier.

Linear Support Vector Machines, SVM determines a hyperplane that separates both classes with maximal margin, given vectors of two classes as training data. One of these classes is associated with malware, whereas the other class corresponds to benign instances. An unknown/new instance is classified by mapping it to the vector space and determining whether it falls on the malicious or benign side of the hyperplane [13]. SVM is widely used in malware classification task as it produces explainable detection model.

Decision Trees, DT builds a rule-based model that predicts the class of a target variable by learning decision rules inferred from the given set of features. The depth of the tree can be customized to fit the model. The deeper the tree, the more complex the decision rules and the fitter the model. Deep trees may not generalize the data well (overfitting problem) and thus, usually it is necessary to limit the maximum depth of the tree. There are a few variants of decision tree such as ID3, C4.5, C5.0, and CART. We use CART [8].

Random Forest, RF is an ensemble of classifiers using many decision tree models [7]. A different subset of training data is selected with a replacement to train each tree. The remaining training data serves to estimate the error and variable importance. RF has been proved to be highly accurate classifier for malware detection [17]. In our experiments, we used 10 classifiers to form an ensemble.

AdaBoost, AB is also an ensemble of classifiers. It fits a sequence of weak models (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote to produce the final classification.

Naive Bayes, NB classifier applies Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable [49]. This assumption allows NB to learn the model extremely fast.

Logistic Regression, LR is a statistical model that uses a logistic function to model the probability of a certain class such as pass/fail, malware/benign, etc.

We used scikit-learn Python tool [32] to run the above classifiers. We used the tool's default settings, such as K=3 for KNN, number of estimators = 10 for RF, etc, without any fine tuning. *Use features* are fed into the classifiers as they are. *Sequence features* are fed into the classifiers as categorical features.

3.3.2 Deep Learning (DL) Classifiers.

Deep learning is a class of machine learning algorithms that uses multiple layers to progressively extract higher level features from the raw input features. Deep learning classifiers typically comprise an input layer, one or more hidden layers, and an output layer. In our context, the input layer accepts vectors of features — *use features* or *sequence features* (Section 3.2). Each vector represents an app. The output layer is the binary classification (benign/malware) of the given app.

This study uses the following deep learning classifiers:

Artificial Neural Network, ANN is a common deep learning method, which comprises of an input layer, one or more hidden, fully-connected (linear) layers, and an output layer. We used two different configurations of ANN — different number of hidden layers and different number of neurons in each layer — in our experiments. The first ANN is a simple ANN, denoted as *sANN*, which consists of two linear layers, with each layer containing 256 neurons. The second ANN is a more complex ANN, denoted as *cANN*. It consists of three linear layers — with 512, 256, 128 neurons, respectively. At the end of these layers, cANN also has a dropout layer with p=0.5 to avoid overfitting [39].

Convolutional Neural Network, CNN typically comprises three types of layers — convolutional layer, pooling layer, and linear layer — between the input layer and the output layer. The convolutional layer utilizes the convolution procedure to accomplish the weight sharing. The pooling layer progressively reduce the dimension of the feature map and thus, reduce the amount of parameters and computation. It can be applied by an average pooling procedure or a max pooling procedure. Thereafter, one or more linear layers and the output layer, typically a SoftMax function, are placed on the top layer for classification and recognition. In our experiments, we built the CNN classifier with the following sequence of layers – the input layer, a convolutional layer followed by a max pooling layer, another convolutional layer, followed by a max pooling layer, and one linear layer, a dropout layer with p=0.5, and finally the output layer with Softmax function.

Recurrent Neural Network, RNN is suitable for handling sequential data. It has memory units, which retain the information of previous inputs or the state of hidden layers and its output depends on previous inputs. It can also have a special layer called LSTM, which avoids the error vanishing problem by fixing weight of hidden layers to avoid error decay and retaining not all information of input but only selected information which is required for future outputs. RNN has shown good results in various fields which use sequential data such as language processing or speech recognition [14]. In our experiments, we built the RNN with LSTM units. Our RNN classifier consists of the input layer, one LSTM layer, one linear layer, and the output layer with Softmax function. We also use dropout with p=0.5.

We built the above DL classifiers by using Pytorch's libraries [31]. As activation function for linear layers and convolutional layers, we use rectified linear unit (ReLU) function. We use 30 epochs for training the DL classifiers. The scripts are written in Python.

Table 1 shows a summary of general comparison among the classifiers we used, based on the documentations from scikit-learn [32] and Pytorch [31], and the references from [28, 29].

Table 1: Pros and cons of the classifiers [28, 29, 31, 32]

Class	Classifier	Pros	Cons
Statistics	Naive Bayes	very fast classifier; suitable for getting quick classification results	unable to learn complex relationships among features
	K-Nearest Neighbours	typically more robust than other statistics-based classifiers for small k value	large memory and computation time for training
	Linear SVM	efficient; easy to analyze output	only directly applicable for binary classification problems; large memory and computation time for training
	Logistic Regression	can learn relatively complex relationships among features	unpredictable performance as the learning process may fail to converge (failure of the likelihood maximization algorithm)
Rules	Random Forest	randomization typically helps achieve good performance	output is hard to analyze
	Decision Trees	fast and scalable classifier; easy to analyze output	less effective when learning features with continuous values
	AdaBoost	built-in feature selection capability, which reduces dimensionality and computation time	sensitive to noisy data and outliers
Deep Learn	Simple/Complex Artificial Neural Network	parelellization of learning process and typically achieves good performance	consume large memory and computation time for both training and classification, compared to typical ML models
	Convolutional Neural Network	fewer neuron connections needed compared to a standard ANN, i.e., faster learning process; can be varied to suit the need to a particular classifier problem	fine tuning is usually needed to discover a complete hierarchy of features; it also needs a big dataset
	Recurrent Neural Network	modeling time dependencies; able to remember serial events	learning process suffers from vanishing gradient problem; fine tuning to suit a given classifier problem is usually needed to avoid this problem

4 EVALUATION

This section presents the experimental comparison results of features and classifiers for Android malware detection. Specifically, we investigate the following research questions:

- **RQ1:** Which type of features — the use of API calls or the sequence of API calls — achieves better malware detection accuracy?
- **RQ2:** Which type of features — statically extracted, dynamically extracted, or a combination of both — achieves better malware detection accuracy?
- **RQ3:** Do deep learning classifiers achieve better malware detection accuracy than conventional machine learning classifiers?
- **RQ4:** What are the training costs for different types of features?

4.1 Experiment Design

Dataset. Initially we had 20k benign samples collected from Androzoo repository [2], which are released from year 2017 to 2019. We also had 7757 malware samples — 5500 samples from Drebin repository [5] and 2257 samples from Androzoo repository [2], which are from year 2017 to 2019. However, as we evaluate the use of both static- and dynamic analysis-based features, we had to filter those samples that can be analyzed by both static analysis and dynamic analysis tools. When we use FlowDroid [6] tool to extract call graphs, some of the apps caused exceptions. And our intent-fuzzing test generation tool also caused time-outs and crashes for some of the apps during the dynamic analysis. Therefore, we were not able to extract features for those cases. Note that these are the limitations of the underlying program analysis tools; for future work, we plan to investigate these issues and address them. Nevertheless, the objective of this experiment is to compare features and classifiers and not to assess the feature collection components.

As a result, after we take the intersection of the apps that can be commonly analyzed by static and dynamic tools, we ended up with 4572 benign samples and 2399 malware samples — 1208 from Androzoo repository [2] and 1191 from Drebin repository [5]. Note that several of the malware samples from Drebin are obfuscated and the malware samples from Androzoo are recent.

In comparison, Table 2 shows the sizes of dataset used by Android malware detection approaches in related work. But note that these studies only apply either static or dynamic analysis and evaluate a few classifiers. Whereas we evaluate 11 classifiers and 6 different types of features. Our dataset size is comparable to the sizes used in some recent studies such as [37, 46].

Table 2: Sizes of dataset used in some of the malware detection approaches

Reference	#Benign	#Malware
Droid-sec [48]	250	250
DroidSift [50]	13500	2200
Drebin [5]	123453	5560
Narudin et al. [28]	20	1000
Maldozer [22]	37627	33066
RevealDroid [19]	24679	30203
Shen et al. [37]	3899	3899
EnMobile [46]	1717	4897
MaMadroid [30]	8447	35493
DaDiDroid [21]	43262	20431

Metrics. To assess the accuracy of the classifiers, we use the standard metrics — Recall (probability of detection, Pd), Precision (Pr), and F-measure (F) — which are typically used for evaluating Malware detection accuracy [19, 30]. Recall is computed as $Pd = tp/(tp + fn)$; Precision is computed as $Pr = tp/(tp + fp)$; and F-measure is computed as $F = 2 * (Pr * Pd)/(Pr + Pd)$.

Given that we have six datasets, our assessment includes six experiments. We use stratified cross validation, a standard statistical analysis method [44], to evaluate the performances. Ten-fold cross validation is widely used [19, 21, 30, 46]. But given that we are evaluating several classifiers and features, we instead used five-fold cross validation, which was also used in [23, 42]. The data is randomly divided into five sets. A classifier is trained on four sets and then tested on the remaining set. This process is repeated five times; each time testing on a different set. The order of training and test set is randomized. This test design overcomes the ordering effects due to randomization. This is important to avoid a malignant increase in performance by a certain ordering of training and test data. Isolating a test set from the training set also conforms to hold-out test design which is important to evaluate the classifier’s capability to predict new malware [44]. The mean and the standard deviation of all five trials is computed to make an evaluation.

The experiments were performed on a Linux machine with 40 cores Intel CPU E5-2640 2.40GHz and 330GB RAM. It took us about a month to extract call graphs and execution traces from all the 27k plus samples. It took us about two weeks to extract the six datasets from the final benchmark set which contains 6971 samples in total.

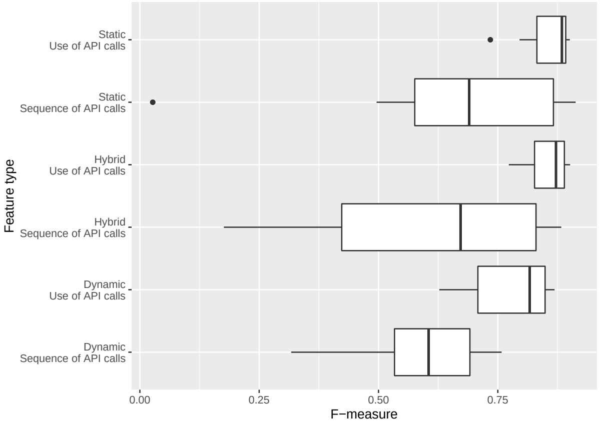
4.2 Result Comparisons

Table 3, Table 4 and Table 5 show the results of classifiers using *static-sequence features, dynamic-sequence features, and hybrid-sequence features,* respectively. Table 6, Table 7 and Table 8 show the results of classifiers using *static-use features, dynamic-use features, and hybrid-use features,* respectively. The columns ‘Pd’, ‘Pr’, ‘F’ represent the mean recall, the mean precision, and the mean F-measure results across cross validations. The columns ‘Pd (sd)’, ‘Pr (sd)’, ‘F (sd)’ represent the standard deviations across cross validations.

4.2.1 RQ1: Use of API calls vs Sequence of API calls. As shown in Tables 3, 4, 5, 6, 7, and 8, the F-measures of classifiers using *use features* are statistically better than those of classifiers using *sequence features* (according to Wilcoxon signed-ranks test). The standard deviations of F-measures for the classifiers using *use features* are generally quite low, with the maximum standard deviation of 0.209 (NB using dynamic API usage features), whereas the standard deviations of F-measures for the classifiers using *sequence features* are statistically higher (according to Wilcoxon signed-ranks test), with the maximum standard deviation of 0.406 (cANN using *static-sequence* features).

In Figure 5, we plot the F-measures achieved by classifiers by using *static-sequence features, dynamic-sequence features, hybrid-sequence features, static-use features, dynamic-use features, and hybrid-use features*, respectively. The figure clearly demonstrates that features which characterize the use of API calls generally produce better results than features which characterize the sequence of API calls across all types of program analyses.

But there are a few exceptions. The best classifier in terms of F-measure is Random Forest, F = 0.913 (Table 3), which is actually trained with *static-sequence features.* AdaBoost also achieved better F-measure scores when trained with *static-sequence* and *hybrid-sequence features.* On the other hand, the second best classifier is sANN, F = 0.901 (Table 8), which is trained with *hybrid-use features.*



Feature type	median	mean	sd
Static - Use of API calls	0.88	0.86	0.06
Static - Sequence of API calls	0.69	0.67	0.26
Dynamic - Use of API calls	0.82	0.77	0.09
Dynamic - Sequence of API calls	0.60	0.60	0.12
Hybrid - Use of API calls	0.87	0.86	0.04
Hybrid - Sequence of API calls	0.67	0.61	0.28

Figure 5: F-measures of different types of features

In general, the results can be considered good given that we use features that are not based on custom (user defined) methods and classes. Approaches that use such features could see a malignant increase in performance because they may then learn on the discriminative features used by the malware. But on the other hand, those approaches may later suffer from class and method renaming obfuscation strategies [5].

Our observation from a few randomly selected malware samples is that API *sequence features* contain more semantic information as they capture a sequence of API calls involved in a malware activity, whereas API *use features* capture malware patterns in a simpler manner — based on what APIs are used in a malware activity.

On the other hand, based on the results, we believe that *sequence features* are generally harder to train with. Parameters of the classifiers such as maximum depth parameter in Decision Tree or Random Forest, K parameter in KNN, number of neurons and memory units in deep learning classifiers, etc. need to be fine tuned to improve the performance. We randomly sampled 100 apps and fine tuned the parameters of some of the classifiers for training with API *sequence features*, and we observed that the accuracy did improve, especially for CNN and RNN deep learning classifiers. Note that the task of optimizing several classifiers on different types of features took a lot of iterations and a lot of resources — training time and computation. Therefore, in this study, we used the default settings for the ML classifiers and we did not fine tune the DL classifiers at all. We leave the task of optimizing the classifiers and systematically evaluating the performance of optimal classifiers as future work.

Overall, we conclude that API *use features* produce better results, on average. That is, classifiers trained with *use features* would detect Android malware with good accuracy and they work out of the box. On the other hand, since API *sequence features* provide more semantic information, if time and effort could be spent on fine tuning the classifier, *sequence features* could be a better choice.

Table 3: Results on using static features that characterize the sequence of API calls at method level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.578	0.109	0.839	0.019	0.683	0.077
SVM	0.588	0.081	0.838	0.015	0.690	0.055
DT	0.885	0.047	0.852	0.039	0.868	0.025
RF	0.920	0.019	0.905	0.027	0.913	0.010
AB	0.903	0.024	0.869	0.027	0.885	0.010
NB	0.989	0.010	0.441	0.027	0.610	0.026
LR	0.843	0.031	0.888	0.036	0.865	0.021
sANN	0.865	0.082	0.837	0.238	0.843	0.105
cANN	0.868	0.188	0.422	0.591	0.496	0.406
CNN	0.667	0.127	0.461	0.196	0.542	0.186
RNN	0.365	0.254	0.014	0.018	0.027	0.034

Table 4: Results on using dynamic features that characterize the sequence of API calls at method level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.651	0.123	0.782	0.106	0.707	0.058
SVM	0.633	0.198	0.785	0.079	0.696	0.130
DT	0.563	0.652	0.722	0.348	0.521	0.397
RF	0.550	0.239	0.691	0.169	0.605	0.171
AB	0.493	0.217	0.698	0.215	0.563	0.121
NB	0.991	0.010	0.376	0.016	0.546	0.016
LR	0.622	0.193	0.777	0.091	0.687	0.140
sANN	0.653	0.024	0.646	0.039	0.649	0.025
cANN	0.699	0.136	0.393	0.094	0.500	0.085
CNN	0.700	0.031	0.828	0.083	0.758	0.035
RNN	0.560	0.147	0.225	0.167	0.317	0.179

Table 5: Results on using hybrid features that characterize the sequence of API calls at method level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.521	0.170	0.803	0.073	0.626	0.102
SVM	0.657	0.133	0.699	0.104	0.672	0.022
DT	0.862	0.086	0.807	0.076	0.832	0.043
RF	0.877	0.081	0.892	0.089	0.883	0.037
AB	0.887	0.088	0.843	0.039	0.863	0.024
NB	0.984	0.024	0.449	0.055	0.616	0.051
LR	0.807	0.148	0.861	0.112	0.828	0.034
sANN	0.899	0.057	0.742	0.243	0.805	0.127
cANN	0.923	0.085	0.135	0.137	0.230	0.199
CNN	0.714	0.134	0.121	0.060	0.204	0.078
RNN	0.527	0.092	0.106	0.018	0.176	0.028

Table 6: Results on using static features that characterize the use of API calls at class level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.617	0.035	0.907	0.041	0.734	0.011
SVM	0.922	0.011	0.863	0.029	0.892	0.01
DT	0.908	0.008	0.833	0.003	0.869	0.002
RF	0.924	0.020	0.879	0.044	0.901	0.028
AB	0.887	0.010	0.847	0.002	0.867	0.006
NB	0.968	0.004	0.675	0.006	0.795	0.005
LR	0.930	0.023	0.872	0.043	0.900	0.012
sANN	0.865	0.023	0.923	0.046	0.893	0.028
cANN	0.865	0.027	0.920	0.045	0.891	0.022
CNN	0.785	0.036	0.810	0.045	0.797	0.032
RNN	0.878	0.023	0.895	0.035	0.884	0.022

Table 7: Results on using dynamic features that characterize the use of API calls at class level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.627	0.253	0.82	0.115	0.706	0.202
SVM	0.839	0.099	0.828	0.095	0.832	0.078
DT	0.660	0.376	0.678	0.270	0.637	0.195
RF	0.807	0.059	0.830	0.139	0.817	0.084
AB	0.723	0.247	0.789	0.096	0.750	0.178
NB	0.857	0.340	0.613	0.146	0.710	0.209
LR	0.848	0.093	0.831	0.117	0.838	0.088
sANN	0.853	0.046	0.887	0.046	0.869	0.016
cANN	0.842	0.031	0.895	0.027	0.868	0.022
CNN	0.638	0.043	0.618	0.062	0.627	0.037
RNN	0.852	0.016	0.869	0.062	0.860	0.027

Table 8: Results on using hybrid features that characterize the use of API calls at class level

Classifier	Pd	Pd (sd)	Pr	Pr (sd)	F	F (sd)
KNN	0.911	0.078	0.833	0.133	0.868	0.064
SVM	0.916	0.056	0.861	0.135	0.885	0.066
DT	0.880	0.138	0.802	0.147	0.833	0.066
RF	0.905	0.074	0.847	0.138	0.872	0.060
AB	0.827	0.229	0.827	0.156	0.820	0.143
NB	0.863	0.272	0.706	0.150	0.773	0.185
LR	0.925	0.067	0.869	0.125	0.894	0.054
sANN	0.873	0.039	0.932	0.045	0.901	0.010
cANN	0.862	0.031	0.935	0.027	0.897	0.016
CNN	0.799	0.023	0.825	0.030	0.812	0.018
RNN	0.879	0.035	0.888	0.037	0.884	0.014

4.2.2 *RQ2: Static vs Dynamic vs Hybrid.*

Referring to Figure 5, we can also observe that the static analysis-based features significantly outperform the dynamic analysis-based features. The mean and median of classifiers using *static-use features* are better than those of classifiers using *dynamic-use features*. The same goes for *static-sequence features* versus *dynamic-sequence features*. The overall standard deviation for dynamic-based features is slightly better than that of static-analysis features.

We looked at our data files and found that the sizes of execution traces are much smaller than the sizes of the call graphs. Our analysis on all the execution traces showed that dynamic analysis was only able to cover 19357 and 163292 distinct standard classes and methods, respectively. By contrast, our analysis on all the call graphs showed that static analysis covered 134558 and 2898245 distinct classes and methods, respectively. Therefore, the static analysis-based features characterize more program behaviours and were more informative for the classifiers than the dynamic analysis-based features.

Note that to cover program behaviors as much as possible, we used an intent fuzzer that handles inter-component (inter-app) interactions complemented with GUI test generator that handles user interaction events and inputs. Even though static analysis is generally weak against code obfuscation and our dataset contains several malware with obfuscated code, this did not have significant effect on static analysis because we only used features that represent standard (not user defined) classes and methods to mitigate renaming obfuscation (see Section 3.2).

Regarding the use of hybrid features, we found that it improved the accuracy for only some of the classifiers and it had the negative effect on other classifiers. For example, KNN's F-measure when trained with *static-use features* is 0.734 and it improved to 0.868 when *static-use* and *dynamic-use* features are combined. The same goes for all four DL classifiers when trained with *use features*. But the result of all other classifiers decreased. With respect to *sequence features*, only NB's F-measure was improved when *static-sequence* and *dynamic-sequence* features are combined. For all the other cases, the F-measure decreased. Therefore, we note that the usefulness of combining static analysis and dynamic analysis is contextual. Since the performances of classifiers using dynamic analysis-based features are generally poor, those features may have polluted the classifiers learning. Hence, we note that for combining statically- and dynamically-extracted information, data preprocessing, such as feature selection to remove redundant or irrelevant features, should be applied.

Overall we conclude that static analysis is more desirable than dynamic analysis, unless we can further improve the state-of-the-art of test generation for Android apps to improve coverage. We find that the limitation of static analysis can be mitigated by focusing on the standard methods and classes. For using hybrid analysis, data preprocessing should be considered.

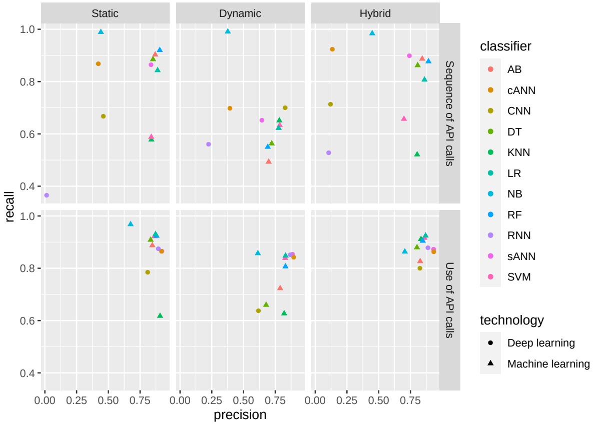
4.2.3 RQ3: ML Classifiers vs DL Classifiers. In Figure 6, we plot the recall and precision grid of the classifiers, averaged across all types of features. Instead of F-measures, we will discuss here the performance of classifiers based on recall and precision. This is because in some contexts, e.g., in highly security-critical systems, a higher recall at the expense of some precision loss is more desirable whereas in some other contexts, a higher precision could be more desirable.

As shown in Figure 6, overall, averaging across all types of features, in terms of both recall and precision, the ML classifiers achieved better scores than the DL classifiers. The ML classifiers achieved the median and mean recall of 0.86 and 0.80, respectively, and achieved the median and mean precision of 0.83 and 0.78, respectively. The DL classifiers achieved the median and mean recall of 0.85 and 0.77, respectively, and achieved the median and mean precision of 0.82 and 0.64, respectively.

One possible reason why the ML classifiers generally perform better than the DL classifiers may be due to the same fine tuning issue as discussed in Section 4.2.1. We used the readily-available ML classifiers from scikit-learn tool [32] with its built-in settings, which may have been optimal for malware detection. By contrast, since there are many different ways to build DL classifiers, Pytorch [31] only provides abstract neural network classes on which custom DL classifiers with specific configurations of DL layers are usually built. As such, without fine tuning the parameters such as the number of layers, the sequence of different types of layers, the number of neurons, dropout value, training epochs, etc., the DL classifiers may not perform well.

On the other hand, the DL classifiers did achieve better results than the ML classifiers when trained with dynamic features, on both API sequence and API usage. As shown in Table 4, CNN achieved Pd = 0.7 and Pr = 0.828, which are better than all other classifiers trained with *dynamic-sequence features*, except NB that has Pd = 0.991 but with Pr = 0.376. Similarly, as shown in Table 7, sANN, cANN, RNN achieved better results than the ML classifiers.

One other interesting finding is that the simplest classifier, Naive Bayes, achieved Pd = 0.942, averaging across all types of features. It is the highest recall among all the classifiers; that is, it detected 2260 out of 2399 malware from our dataset. On the other hand, it only achieved Pr = 0.543 on average; that is, it produced one false alarm for every two malware reports. When detecting malware is of utmost importance, Naive Bayes can be a good option. When better precision is more desirable, RF can be considered, which achieved Pd = 0.831 and Pr = 0.841, averaging across all types of features. It is the most balanced classifier and thus, can be considered as the best, at least in our experiments.



Classifier type	Pd (median)	Pd (mean)	Pd (sd)
Machine learning	0.86	0.80	0.15
Deep learning	0.85	0.77	0.14
	Pr (median)	Pr (mean)	Pr (sd)
Machine learning	0.83	0.78	0.12
Deep learning	0.82	0.64	0.32

Figure 6: Recall and precision of classifiers

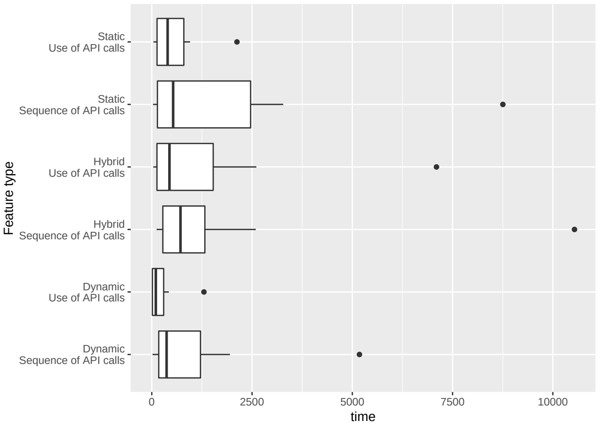
4.2.4 RQ4: Training Costs. As Android platform is constantly evolving, a malware detector may often need to be re-trained to learn the new characteristics of Android. A slow training and analysis of Android apps could allow malware to remain undetected long enough and cause undesirable effects on end users. Hence it is important to assess the training cost of classifiers on using different types of features.

In Figure 7, we plot the time taken by the classifiers when training with different types of features. The figure illustrates the average time taken for training the classifiers on each type of features. In our five fold cross validation setting, the training time is computed as the time taken to train on the four folds of the dataset. We first compare the training time required for static, dynamic, and hybrid cases. We then compare the training time required for API usage features versus API sequence features.

As shown in Figure 7, intuitively, since there are more number of features for hybrid case, classifiers took the longest to train with hybrid features. Comparing the static and dynamic cases, classifiers using the static-based features took longer than those using the dynamic-based features. On average, the training cost of using static-based features (1141 seconds) is 1.78 times more than the training cost of using dynamic-based features (641 seconds).

We can also observe from Figure 7 that API *use features* are much faster to train with than API *sequence features.* On average, the training cost of using API *sequence features* (1485 seconds) is two times more than that of using API *use features* (719 seconds).

Hence, overall it can be concluded that while dynamic-based features are less accurate, they are much faster to train with, compared to static-based features. API *use features* are both faster and simpler (no fine tuning required to achieve good accuracy) to train with, compared to API *sequence features.*



Feature type	median	mean	sd
Static - Use of API calls	395.01	580.36	614.93
Static - Sequence of API calls	532.42	1700.96	2624.58
Dynamic - Use of API calls	100.45	243.83	379.08
Dynamic - Sequence of API calls	369.50	1038.47	1514.59
Hybrid - Use of API calls	440.87	1333.75	2095.63
Hybrid - Sequence of API calls	716.40	1715.60	3018.26

Figure 7: Training time (in seconds) for different types of features

4.3 Limitations

The main limitation of this work is that our study excludes fine-tuning the parameters or data preprocessing except specifying API sequence features as categorical. Tuning the parameters on the eleven classifiers and the six types of features we used would require huge amount of time and resources. Therefore, this study reports the malware detection accuracy of baseline classifiers, without being optimized. Hence, researchers are to consider the results regarding the performances of classifiers as one data point, a starting point for further exploration of optimized classifiers. This is the subject of our future work. Especially, based on our current results, we observed that this limitation hurts the deep learning classifiers more. This could be the main focus of our future plan. In addition, it would also be interesting to investigate if applying data preprocessing such as feature selection would result in better performance for hybrid features.

Our dataset is imbalanced. Our dataset's benign-to-malware ratio is 1.9 to 1, which may, in theory, affect precision and recall. However, while it is challenging to approximate the actual ratio of benign apps versus malware apps in the wild, it is more likely that they are not balanced. Some study has chosen imbalanced dataset [5, 21] and some has chosen balanced dataset [23, 37]. Certain Android markets have been known to have a benign-to-malware ratio of 1.5 to 1 [10]. Hence, our dataset could reflect the reality better. We plan to investigate the implication of different dataset ratios in future.

Our analysis does not consider native calls although FlowDroid, the underlying static analysis tool we use, handles common native calls using some heuristics. It is a challenging task to extract features that characterize native calls using static analysis. Dynamic runtime analysis approaches such as [15, 41] could be used. Our study also did not consider API calls *frequency.* A recent study [30] found that their proposed malware detection model is less accurate when trained on API calls frequency features instead of API sequence features. We plan to include this evaluation in our future work.

5 CONCLUSION

In this work, we evaluated six different types of features and eleven classifiers. The features characterize the use of API calls at class level and the sequence of API calls at method level. Both static analysis and dynamic analysis are used. The classifiers include both conventional machine learning and deep learning models. To assess the accuracy, recall, precision, and mainly F-measure were used. We also discussed the training costs. The experiments were conducted on a common benchmark, containing 4572 benign samples and 2399 malware samples.

Our results show that compared to the features which characterize the sequence of API calls, the features which characterize the use of API calls are faster and simpler to train with and produce classifiers with better accuracies in general. Static analysis-based features characterize more program behaviours compared to dynamic analysis-based features. Hence, they produced classifiers with better accuracies but they came with training cost which is 1.78 times longer on average. Overall, the best F-measure (0.913) was achieved by a ML classifier, Random Forest classifier, which was trained with the static API sequence-based features. The second best F-measure (0.901) was achieved by a DL classifier, a simple Artificial Neural Network model, which was trained with the hybrid API usage-based features. In our future work, we plan to investigate into data preprocessing, feature selection, and parameter fine tuning to produce optimal classifiers and evaluate their impacts. We also plan to evaluate *frequency*-based features.

ACKNOWLEDGMENT

The work of L. K. Shar is supported by the National Research Foundation Singapore, under the National Satellite of Excellence in Mobile System Security and Cloud Security (NRF2018NCR-NSOE004-0001).

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 468–471. ACM, 2016.

[3] H. Alshahrani, H. Mansourt, S. Thorn, A. Alshehri, A. Alzahrani, and H. Fu. Ddefender: Android application threat detection using static and dynamic analysis. In *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6. IEEE, 2018.

[4] Android. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey, 2019.

[5] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[7] I. Barandiaran. The random subspace method for constructing decision forests. *IEEE Trans. Pattern Anal. Mach. Intell*, 20(8):1–22, 1998.

[8] L. Breiman. *Classification and regression trees*. Routledge, 2017.

[9] P. P. Chan and W.-K. Song. Static detection of android malware by using permissions and api calls. In *2014 International Conference on Machine Learning and Cybernetics*, volume 1, pages 82–87. IEEE, 2014.

[10] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 659–674, 2015.

[11] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388, 2016.

[12] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46. IEEE, 2005.

[13] N. Cristianini, J. Shawe-Taylor, et al. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[14] L. Deng, D. Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[15] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 240–253. Springer, 2012.

[16] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[17] M. Eskandari and S. Hashemi. A graph mining approach for detecting unknown malwares. *Journal of Visual Languages & Computing*, 23(3):154–162, 2012.

[18] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu. Frequent subgraph based familial classification of android malware. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–35. IEEE, 2016.

[19] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[20] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu. Performance evaluation on permission-based detection for android malware. In *Advances in Intelligent Systems and Applications-Volume 2*, pages 111–120. Springer, 2013.

[21] M. Ikram, P. Beaume, and M. A. Kaafar. Dadidroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling. *arXiv preprint arXiv:1905.09136*, 2019.

[22] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb. Android malware detection using deep learning on api method sequences. *arXiv preprint arXiv:1712.08996*, 2017.

[23] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb. Maldozer: Automatic framework for android malware detection using deep learning. *Digital Investigation*, 24:S48–S59, 2018.

[24] Y. Liao and V. R. Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & security*, 21(5):439–448, 2002.

[25] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*, pages 3–17. IEEE, 2014.

[26] X. Liu and J. Liu. A two-layered permission-based android malware detection scheme. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 142–148. IEEE, 2014.

[27] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308. ACM, 2017.

[28] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[29] A. Naway and Y. Li. A review on the use of deep learning in android malware detection. *arXiv preprint arXiv:1812.10360*, 2018.

[30] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):14, 2019.

[31] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[32] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[33] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334, 2013.

[34] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISISâĂŹ12-ICEUTE 12-SOCO 12 Special Sessions*, pages 289–298. Springer, 2013.

[35] L. K. Shar. Experimental comparison of features and machine learning classifiers for android malware detection. https://github.com/sharlwinkhin/msoft20, 2020.

[36] A. Sharma and S. K. Dash. Mining api calls and permissions for android malware detection. In *International Conference on Cryptology and Network Security*, pages 191–205. Springer, 2014.

[37] F. Shen, J. Del Vecchio, A. Mohaisen, S. Y. Ko, and L. Ziarek. Android malware detection using complex-flows. *IEEE Transactions on Mobile Computing*, 18(6):1231–1245, 2018.

[38] Soot. Soot - a java optimization framework, https://github.com/sable/soot. 2018.

[39] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[40] Symantec. Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-24-2019-en.pdf, 2019.

[41] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582. IEEE, 2016.

[42] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi. Malware detection with deep neural network using process behavior. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 577–582. IEEE, 2016.

[43] R. Vinayakumar, K. Soman, P. Poornachandran, and S. Sachin Kumar. Detecting android malware using long short-term memory (lstm). *Journal of Intelligent & Fuzzy Systems*, 34(3):1277–1288, 2018.

[44] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[45] K. Xu, Y. Li, R. H. Deng, and K. Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 473–487. IEEE, 2018.

[46] W. Yang, M. Prasad, and T. Xie. Enmobile: Entity-based characterization and analysis of mobile malware. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 384–394. IEEE, 2018.

[47] S. Y. Yerima, S. Sezer, and I. Muttik. High accuracy android malware detection using ensemble learning. *IET Information Security*, 9(6):313–320, 2015.

[48] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[49] H. Zhang. The optimality of naive bayes. *AA*, 1(2):3, 2004.

[50] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116, 2014.